ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
UIT

GIÁO TRÌNH

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Biên soạn: Trần Hạnh Nhi
Dương Anh Đức

Thư viện ĐH Trà Vinh
1102346

3
00
010

NHÀ XUẤT BẢN
ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

Giáo trình

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Biên soạn : TRẦN HẠNH NHI
DƯƠNG ANH ĐỨC

TRƯỜNG ĐẠI HỌC TRÀ VINH
THƯ VIỆN
PHÒNG MƯỢN

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA
THÀNH PHỐ HỒ CHÍ MINH - 2010

LỜI NÓI ĐẦU

Giáo trình này là một trong các giáo trình chính yếu của chuyên ngành Công nghệ thông tin. Giáo trình được xây dựng theo phương châm vừa đáp ứng yêu cầu chuẩn mực của sách giáo khoa, vừa có giá trị thực tiễn, đồng thời tăng cường khả năng tự học, tự nghiên cứu của sinh viên. Trên cơ sở đó, chúng tôi đã tham khảo nhiều tài liệu có giá trị của các tác giả trong và ngoài nước và đã sử dụng nhiều ví dụ lấy từ các ứng dụng thực tiễn.

Giáo trình này được dùng kèm giáo trình điện tử trên đĩa CD trong đó có thêm phần trình bày của giảng viên , các bài tập và phần đọc thêm nhằm đáp ứng tốt nhất cho việc tự học của sinh viên.

Chúng tôi rất mong nhận được các ý kiến đóng góp để giáo trình ngày càng hoàn thiện.

Nhóm biên soạn

CHƯƠNG 1

TỔNG QUAN VỀ GIẢI THUẬT VÀ CẤU TRÚC DỮ LIỆU

Mục tiêu

- Giới thiệu vai trò của việc tổ chức dữ liệu trong một đề án tin học.
- Mối quan hệ giữa giải thuật và cấu trúc dữ liệu.
- Các yêu cầu tổ chức cấu trúc dữ liệu
- Khái niệm kiểu dữ liệu_cấu trúc dữ liệu
- Tổng quan về đánh giá độ phức tạp giải thuật

I. VAI TRÒ CỦA CẤU TRÚC DỮ LIỆU TRONG MỘT ĐỀ ÁN TIN HỌC

- Thực hiện một đề án tin học là chuyển bài toán thực tế thành bài toán có thể giải quyết trên máy tính. Một bài toán thực tế bất kỳ đều bao gồm các đối tượng dữ liệu và các yêu cầu xử lý trên những đối tượng đó. Vì thế, để xây dựng một mô hình tin học phản ánh được bài toán thực tế cần chú trọng đến hai vấn đề :

Tổ chức biểu diễn các đối tượng thực tế

Các thành phần dữ liệu thực tế đa dạng, phong phú và thường chứa đựng những quan hệ nào đó với nhau, do đó trong mô hình tin học của bài toán, cần phải tổ chức , xây

dựng các cấu trúc thích hợp nhất sao cho vừa có thể phản ánh chính xác các dữ liệu thực tế này, vừa có thể dễ dàng dùng máy tính để xử lý. Công việc này được gọi là xây dựng *cấu trúc dữ liệu* cho bài toán.

Xây dựng các thao tác xử lý dữ liệu

Từ những yêu cầu xử lý thực tế, cần tìm ra các giải thuật tương ứng để xác định trình tự các thao tác máy tính phải thi hành để cho ra kết quả mong muốn, đây là bước xây dựng *giải thuật* cho bài toán.

Tuy nhiên khi giải quyết một bài toán trên máy tính, chúng ta thường có khuynh hướng chỉ chú trọng đến việc xây dựng giải thuật mà quên đi tầm quan trọng của việc tổ chức dữ liệu trong bài toán. Giải thuật phản ánh các phép xử lý , còn đối tượng xử lý của giải thuật lại là dữ liệu, chính dữ liệu chứa đựng các thông tin cần thiết để thực hiện giải thuật. Để xác định được giải thuật phù hợp cần phải biết nó tác động đến loại dữ liệu nào (ví dụ để làm nhuyễn các hạt đậu, người ta dùng cách xay chứ không băm bằng dao, vì đậu sẽ văng ra ngoài) và khi chọn lựa cấu trúc dữ liệu cũng cần phải hiểu rõ những thao tác nào sẽ tác động đến nó (ví dụ để biểu diễn các điểm số của sinh viên người ta dùng số thực thay vì chuỗi ký tự vì còn phải thực hiện thao tác tính trung bình từ những điểm số đó). Như vậy trong một đề án tin học, giải thuật và cấu trúc dữ liệu có mối quan hệ chặt chẽ với nhau, được thể hiện qua công thức :

Cấu trúc dữ liệu + Giải thuật = Chương trình

Với một cấu trúc dữ liệu đã chọn, sẽ có những giải thuật tương ứng, phù hợp. Khi cấu trúc dữ liệu thay đổi, thường giải thuật cũng phải thay đổi theo để tránh việc xử lý gượng ép, thiếu tự

nhiên trên một cấu trúc không phù hợp. Hơn nữa, một cấu trúc dữ liệu tốt sẽ giúp giải thuật xử lý trên đó có thể phát huy tác dụng tốt hơn, vừa đáp ứng nhanh vừa tiết kiệm vật tư, giải thuật cũng dễ hiểu và đơn giản hơn.

Ví dụ 1: Một chương trình quản lý điểm thi của sinh viên cần lưu trữ các điểm số của 3 sinh viên. Do mỗi sinh viên có 4 điểm số ứng với 4 môn học khác nhau nên dữ liệu có dạng bảng như sau:

Sinh viên	Môn 1	Môn 2	Môn 3	Môn 4
SV 1	7	9	5	2
SV 2	5	0	9	4
SV 3	6	3	7	4

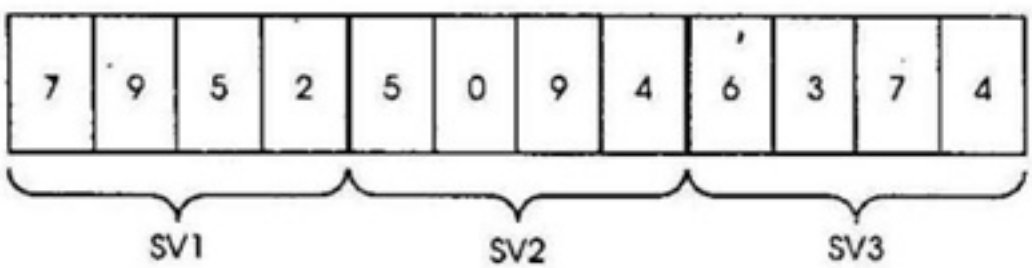
Chỉ xét thao tác xử lý là xuất điểm số các môn của từng sinh viên. Giả sử có các phương án tổ chức lưu trữ sau:

Phương án 1 : Sử dụng mảng một chiều

Có tất cả 3(SV)*4(Môn) = 12 điểm số cần lưu trữ, do đó khai báo mảng *result* như sau :

```
int   result [ 12 ] =   {7, 9, 5, 2,
                         5, 0, 9, 4,
                         6, 3, 7, 4};
```

khi đó trong mảng *result* các phần tử sẽ được lưu trữ như sau:



Và truy xuất điểm số môn j của sinh viên i - là phần tử tại (dòng i, cột j) trong bảng - phải sử dụng một công thức xác định chỉ số tương ứng trong mảng result:

bảngđiểm(dòng i, cột j) ⇒ result[((i-1)*số cột) + j]

Ngược lại, với một phần tử bất kỳ trong mảng, muốn biết đó là điểm số của sinh viên nào, môn gì, phải dùng công thức xác định sau

result[i] ⇒ bảngđiểm (dòng((i / số cột) +1), cột (i % số cột))

Với phương án này, thao tác xử lý được cài đặt như sau :

```
void  XuatDiem()        //Xuất điểm số của tất cả sinh viên
{
   const int   so_mon = 4;
   int   sv,mon;
   for (int i=0; i<12; i+)
   {
      sv = i/so_mon;   mon = i % so_mon;
      printf("Điểm môn %d của sv %d là: %d", mon, sv,
               result[i]);
   }
}
```

Phương án 2 : Sử dụng mảng 2 chiều

Khai báo mảng 2 chiều *result* có kích thước 3 dòng* 4 cột như sau :

```
int   result[3][4] ={{ 7, 9, 5, 2},
                     { 5, 0, 9, 4},
                     { 6, 3, 7, 4 }};
```

khi đó trong mảng *result* các phần tử sẽ được lưu trữ như sau :

	Cột 0	Cột 1	Cột 2	Cột 3
Dòng 0	result[0][0] =7	result[0][1] =9	result[0][2] =5	result[0][3] =2
Dòng 1	result[1][0] =5	result[1][1] =0	result[1][2] =9	result[1][3] =4
Dòng 2	result[2][0] =6	result[2][1] =3	result[2][2] =7	result[2][3] =4

Và truy xuất điểm số môn j của sinh viên i - là phần tử tại (dòng i, cột j) trong bảng - cũng chính là phần tử nằm ở vị trí (dòng i, cột j) trong mảng

bảngđiểm(dòng i,cột j) ⇒ result[i] [j]

Với phương án này, thao tác xử lý được cài đặt như sau :

```
void  XuatDiem()        //Xuất điểm số của tất cả sinh viên
{
   int   so_mon = 4, so_sv =3;

   for ( int i=0; i<so_sv; i+)
      for ( int j=0; i<so_mon; j+)
      printf("Điểm môn %d của sv %d là: %d", j, i,
               result[i][j]);
}
```

✍ NHẬN XÉT

Có thể thấy rõ phương án 2 cung cấp một cấu trúc lưu trữ phù hợp với dữ liệu thực tế hơn phương án 1, và do vậy giải thuật xử lý trên cấu trúc dữ liệu của phương án 2 cũng đơn giản, tự nhiên hơn.

II. CÁC TIÊU CHUẨN ĐÁNH GIÁ CẤU TRÚC DỮ LIỆU

Do tầm quan trọng đã được trình bày trong phần 1.1, nhất thiết phải chú trọng đến việc lựa chọn một phương án tổ chức dữ liệu thích hợp cho để án. Một cấu trúc dữ liệu tốt phải thỏa mãn các tiêu chuẩn sau :

* *Phản ánh đúng thực tế :* Đây là tiêu chuẩn quan trọng nhất, quyết định tính đúng đắn của toàn bộ bài toán. Cần xem xét kỹ lưỡng cũng như dự trù các trạng thái biến đổi của dữ liệu trong chu trình sống để có thể chọn cấu trúc dữ liệu lưu trữ

thể hiện chính xác đối tượng thực tế.

Ví dụ : Một số tình huống chọn cấu trúc lưu trữ sai :

- Chọn một biến số nguyên **int** để lưu trữ tiền thưởng bán hàng (được tính theo công thức tiền thưởng bán hàng = trị giá hàng * 5%), do vậy sẽ làm tròn mọi giá trị tiền thưởng gây thiệt hại cho nhân viên bán hàng. Trường hợp này phải sử dụng biến số thực để phản ánh đúng kết quả của công thức tính thực tế.

- Trong trường trung học, mỗi lớp có thể nhận tối đa 28 học sinh. Lớp hiện có 20 học sinh, mỗi tháng mỗi học sinh đóng học phí $10. Chọn một biến số nguyên **unsigned char** (khả năng lưu trữ 0 - 255) để lưu trữ tổng học phí của lớp học trong tháng, nếu xảy ra trường hợp có thêm 6 học sinh được nhận vào lớp thì giá trị tổng học phí thu được là $260, vượt khỏi khả năng lưu trữ của biến đã chọn, gây ra tình trạng tràn, sai lệch.

* *Phù hợp với các thao tác trên đó*: Tiêu chuẩn này giúp tăng tính hiệu quả của đề án: việc phát triển các thuật toán đơn giản, tự nhiên hơn; chương trình đạt hiệu quả cao hơn về tốc độ xử lý.

Ví dụ: Một tình huống chọn cấu trúc lưu trữ không phù hợp:

Cần xây dựng một chương trình soạn thảo văn bản, các thao tác xử lý thường xảy ra là chèn, xoá sửa các ký tự trên văn bản. Trong thời gian xử lý văn bản, nếu chọn cấu trúc lưu trữ văn bản trực tiếp lên tập tin thì sẽ gây khó khăn khi xây

dựng các giải thuật cập nhật văn bản và làm chậm tốc độ xử lý của chương trình vì phải làm việc trên bộ nhớ ngoài. Trường hợp này nên tìm một cấu trúc dữ liệu có thể tổ chức ở bộ nhớ trong để lưu trữ văn bản suốt thời gian soạn thảo.

! LƯU Ý

☞ Đối với mỗi ứng dụng , cần chú ý đến thao tác nào được sử dụng nhiều nhất để lựa chọn cấu trúc dữ liệu cho thích hợp.

* *Tiết kiệm tài nguyên hệ thống:* Cấu trúc dữ liệu chỉ nên sử dụng tài nguyên hệ thống vừa đủ để đảm nhiệm được chức năng của nó. Thông thường có 2 loại tài nguyên cần lưu tâm nhất : CPU và bộ nhớ. Tiêu chuẩn này nên cân nhắc tùy vào tình huống cụ thể khi thực hiện đề án . Nếu tổ chức sử dụng đề án cần có những xử lý nhanh thì khi chọn cấu trúc dữ liệu yếu tố tiết kiệm thời gian xử lý phải đặt nặng hơn tiêu chuẩn sử dụng tối ưu bộ nhớ, và ngược lại.

Ví dụ : Một số tình huống chọn cấu trúc lưu trữ lãng phí:

- Sử dụng biến **int** (2 bytes) để lưu trữ một giá trị cho biết tháng hiện hành . Biết rằng tháng chỉ có thể nhận các giá trị từ 1-12, nên chỉ cần sử dụng kiểu **char** (1 byte) là đủ.

- Để lưu trữ danh sách học viên trong một lớp, sử dụng mảng 50 phần tử (giới hạn số học viên trong lớp tối đa là 50). Nếu số lượng học viên thật sự ít hơn 50, thì gây lãng phí. Trường hợp này cần có một cấu trúc dữ liệu linh động hơn mảng- ví dụ xâu liên kết - sẽ được bàn đến trong các chương sau.

III. KIỂU DỮ LIỆU

Máy tính thực sự chỉ có thể lưu trữ dữ liệu ở dạng nhị phân thô sơ. Nếu muốn phản ánh được dữ liệu thực tế đa dạng và phong phú,cần phải xây dựng những phép ánh xạ, những qui tắc tổ chức phức tạp che lên tầng dữ liệu thô, nhằm đưa ra những khái niệm logic về hình thức lưu trữ khác nhau thường được gọi là *kiểu dữ liệu* . Như đã phân tích ở phần 1.1, giữa hình thức lưu trữ dữ liệu và các thao tác xử lý trên đó có quan hệ mật thiết với nhau. Từ đó có thể đưa ra một định nghĩa cho kiểu dữ liệu như sau :

1. Định nghĩa kiểu dữ liệu

Kiểu dữ liệu **T** được xác định bởi một bộ **<V,O>** , với :

- **V : tập các giá trị hợp lệ mà một đối tượng kiểu T có thể lưu trữ**
- **O : tập các thao tác xử lý có thể thi hành trên đối tượng kiểu T.**

Ví dụ: Giả sử có kiểu dữ liệu **mẫu tự** = <V_c ,O_c> với

V_c = { a-z,A-Z}
O_c = { lấy mã ASCII của ký tự, biến đổi ký tự thường thành ký tự hoa...}

Giả sử có kiểu dữ liệu **số nguyên** = <Vi ,Oi> với

Vi = { -32768..32767}
O_i = { +, -, *, /, %}

Như vậy, muốn sử dụng một kiểu dữ liệu cần nắm vững cả nội

dụng dữ liệu được phép lưu trữ và các xử lý tác động trên đó.

Các thuộc tính của 1 KDL bao gồm:

- Tên KDL
- Miền giá trị
- Kích thước lưu trữ
- Tập các toán tử tác động lên KDL

2. Các kiểu dữ liệu cơ bản

Các loại dữ liệu cơ bản thường là các loại dữ liệu đơn giản, không có cấu trúc. Chúng thường là các giá trị vô hướng như các số nguyên, số thực, các ký tự, các giá trị logic ... Các loại dữ liệu này, do tính thông dụng và đơn giản của mình, thường được các ngôn ngữ lập trình (NNLT) cấp cao xây dựng sẵn như một thành phần của ngôn ngữ để giảm nhẹ công việc cho người lập trình. Chính vì vậy đôi khi người ta còn gọi chúng là các kiểu dữ liệu định sẵn.

Thông thường, các kiểu dữ liệu cơ bản bao gồm :

- **Kiểu có thứ tự rời rạc: số nguyên, ký tự, logic , liệt kê, miền con ...**
- **Kiểu không rời rạc: số thực**

Tùy ngôn ngữ lập trình, các kiểu dữ liệu định nghĩa sẵn có thể khác nhau đôi chút. Với ngôn ngữ C, các kiểu dữ liệu này chỉ gồm số nguyên, số thực, ký tự. Và theo quan điểm của C, kiểu ký tự thực chất cũng là kiểu số nguyên về mặt lưu trữ, chỉ khác về cách sử dụng. Ngoài ra, giá trị logic ĐÚNG (TRUE) và giá trị logic SAI

(FALSE) được biểu diễn trong C như là các giá trị nguyên khác zero và zero. Trong khi đó PASCAL định nghĩa tất cả các kiểu dữ liệu cơ sở đã liệt kê ở trên và phân biệt chúng một cách chặt chẽ. Trong giới hạn giáo trình này, ngôn ngữ chính dùng để minh họa sẽ là C.

Các kiểu dữ liệu định sẵn trong C gồm các kiểu sau:

Tên kiểu	Kthước	Miền giá trị	Ghi chú
Char	01 byte	-128 đến 127	Có thể dùng như số nguyên 1 byte có dấu hoặc kiểu ký tự
unsign char	01 byte	0 đến 255	Số nguyên 1 byte không dấu
int	02 byte	-32738 đến 32767	
unsign int	02 byte	0 đến 65335	Có thể gọi tắt là unsign
long	04 byte	-2^{32} đến 2^{31} -1	
unsign long	04 byte	0 đến 2^{32}-1	
float	04 byte	3.4E-38 ... 3.4E38	Giới hạn chỉ trị tuyệt đối. Các giá trị <3.4E-38 được coi = 0. Tuy nhiên kiểu float chỉ có 7 chữ số có nghĩa.
double	08 byte	1.7E-308 ...1.7E308	
long double	10 byte	3.4E-4932...1.1E4932	

Một số điều đáng lưu ý đối với các kiểu dữ liệu cơ bản trong C là kiểu ký tự (char) có thể dùng theo hai cách (số nguyên 1 byte hoặc ký tự). Ngoài ra C không định nghĩa kiểu logic (boolean) mà nó đơn giản đồng nhất một giá trị nguyên khác 0 với giá trị TRUE và giá trị 0 với giá trị FALSE khi có nhu cầu xét các giá trị logic.

Như vậy, trong C xét cho cùng chỉ có 2 loại dữ liệu cơ bản là số nguyên và số thực; tức là chỉ có dữ liệu số. Hơn nữa các số nguyên trong C có thể được thể hiện trong 3 hệ cơ số là hệ thập phân, hệ thập lục phân và hệ bát phân. Nhờ những quan điểm trên, C rất được những người lập trình chuyên nghiệp thích dùng.

Các kiểu cơ sở rất đơn giản và không thể hiện rõ sự tổ chức dữ liệu trong một cấu trúc, thường chỉ được sử dụng làm nền để xây dựng các kiểu dữ liệu phức tạp khác.

3. Các kiểu dữ liệu có cấu trúc

Tuy nhiên trong nhiều trường hợp, chỉ với các kiểu dữ liệu cơ sở không đủ để phản ánh tự nhiên và đầy đủ bản chất của sự vật thực tế, dẫn đến nhu cầu phải xây dựng các kiểu dữ liệu mới dựa trên việc tổ chức, liên kết các thành phần dữ liệu có kiểu dữ liệu đã được định nghĩa. Những kiểu dữ liệu được xây dựng như thế gọi là kiểu dữ liệu có cấu trúc. Đa số các ngôn ngữ lập trình đều cài đặt sẵn một số kiểu có cấu trúc cơ bản như mảng, chuỗi, tập tin, bản ghi...và cung cấp cơ chế cho lập trình viên tự định nghĩa kiểu dữ liệu mới.

Ví dụ: Để mô tả một đối tượng sinh viên, cần quan tâm đến các thông tin sau:

- Mã sinh viên: chuỗi ký tự
- Tên sinh viên: chuỗi ký tự
- Ngày sinh: kiểu ngày tháng
- Nơi sinh: chuỗi ký tự
- Điểm thi: số nguyên

Các kiểu dữ liệu cơ sở cho phép mô tả một số thông tin như :

```
int     Diemthi;
```

Các thông tin khác đòi hỏi phải sử dụng các kiểu có cấu trúc như :

```
char    masv[15];
char    tensv[15];
char    noisinh[15];
```

Để thể hiện thông tin về ngày tháng năm sinh cần phải xây dựng một kiểu bản ghi,

```
typedef struct  tagDate{
   char  ngay;
   char  thang;
   char  thang;
}Date;
```

Cuối cùng, ta có thể xây dựng kiểu dữ liệu thể hiện thông tin về một sinh viên :

```
typedef struct  tagSinhVien{
   char  masv[15];
   char  tensv[15];
   char  noisinh[15];
   int   Diem thi;
}SinhVien;
```

Giả sử đã có cấu trúc phù hợp để lưu trữ một sinh viên, nhưng thực tế lại cần quản lý nhiều sinh viên, lúc đó nảy sinh nhu cầu xây dựng kiểu dữ liệu mới... Mục tiêu của việc nghiên cứu cấu trúc dữ liệu chính là tìm những phương cách thích hợp để tổ chức, liên kết dữ liệu, hình thành các kiểu dữ liệu có cấu trúc từ những kiểu dữ liệu đã được định nghĩa.

4. Một số kiểu dữ liệu có cấu trúc cơ bản

a. Kiểu chuỗi ký tự

Chuỗi ký tự là một trong các kiểu dữ liệu có cấu trúc đơn giản nhất và thường các ngôn ngữ lập trình đều định nghĩa nó như một

kiểu cơ bản. Do tính thông dụng của kiểu chuỗi ký tự các ngôn ngữ lập trình luôn cung cấp sẵn một bộ các hàm thư viện các xử lý trên kiểu dữ liệu này. Đặc biệt trong C thư viện các hàm xử lý chuỗi ký tự rất đa dạng và phong phú. Các hàm này được đặt trong thư viện **string.lib** của C.

Chuỗi ký tự trong C được cấu trúc như một chuỗi liên tiếp các ký tự kết thúc bằng ký tự có mã ASCII bằng 0 (NULL character). Như vậy, giới hạn chiều dài của một chuỗi ký tự trong C là 1 Segment (tối đa chứa 65335 ký tự), ký tự đầu tiên được đánh số là ký tự thứ 0.

TRƯỜNG ĐẠI HỌC TRÀ VINH
THƯ VIỆN
PHÔNG MUỘN

Ta có thể khai báo một chuỗi ký tự theo một số cách sau đây:

```
char   S[10];      // Khai báo một chuỗi ký tự S có chiều dài
                   // tối đa 10 (kể cả kí tự kết thúc)
char   S[]="ABC";// Khai báo một chuỗi ký tự S có chiều
                   // dài bằng chiều dài của chuỗi "ABC"
                   // và giá trị khởi đầu của S là "ABC"
char   *S ="ABC";// Giống cách khai báo trên.
```

Trong ví dụ trên ta cũng thấy được một hằng chuỗi ký tự được thể hiện bằng một chuỗi ký tự đặt trong cặp ngoặc kép "".

Các thao tác trên chuỗi ký tự rất đa dạng. Sau đây là một số thao tác thông dụng:

- So sánh 2 chuỗi: **strcmp**
- Sao chép 2 chuỗi: **strcpy**
- Kiểm tra 1 chuỗi nằm trong chuỗi kia: **strstr**
- Cắt 1 từ ra khỏi 1 chuỗi: **strtok**
- Đổi 1 số ra chuỗi: **itoa**

- Đổi 1 chuỗi ra số: **atoi, atof, ...**
- Đổi 1 hay 1 số giá trị ra chuỗi: **sprintf**
- Nhập một chuỗi: **gets**
- Xuất một chuỗi: **puts**

b. Kiểu mảng

Kiểu dữ liệu mảng là kiểu dữ liệu trong đó mỗi phần tử của nó là một tập hợp có thứ tự các giá trị có cùng cấu trúc được lưu trữ liên tiếp nhau trong bộ nhớ. Mảng có thể một chiều hay nhiều chiều. Một dãy số chính là hình tượng của mảng 1 chiều, ma trận là hình tượng của mảng 2 chiều.

Một điều đáng lưu ý là mảng 2 chiều có thể coi là mảng một chiều trong đó mỗi phần tử của nó là 1 mảng một chiều. Tương tự như vậy, một mảng n chiều có thể coi là mảng 1 chiều trong đó mỗi phần tử là 1 mảng n-1 chiều.

Hình tượng này được thể hiện rất rõ trong cách khai báo của C.

Mảng 1 chiều được khai báo như sau:

<Kiểu dữ liệu> <Tên biến>[<Số phần tử>];

Ví dụ: Để khai báo một biến có tên a là một mảng nguyên 1 chiều có tối đa 100 phần tử ta phải khai báo như sau:

```
int     a[100];
```

Ta cũng có thể vừa khai báo vừa gán giá trị khởi động cho một mảng như sau:

```
int     a[5] = (1, 7, -3, 8, 19);
```

Trong trường hợp này C cho phép ta khai báo một cách tiện lợi hơn

```
int     a[] = (1, 7, -3, 8, 19);
```

Như ta thấy, ta không cần chỉ ra số lượng phần tử cụ thể trong khai báo. Trình biên dịch của C sẽ tự động làm việc này cho chúng ta.

Tương tự, ta có thể khai báo một mảng 2 chiều hay nhiều chiều theo cú pháp sau:

<Kiểu dữ liệu> <Tên biến>[<Số phần tử1>][<Số phần tử2>]...;

Ví dụ, ta có thể khai báo:

```
int   a[100][150];
```

hay

```
int   a[][]={{1, 7, -3, 8, 19},
             {4, 5, 2, 8, 9},
             {21, -7, 45, -3, 4}};
```

(mảng a sẽ có kích thước là 3x5).

Các thao tác trên mảng 1 chiều sẽ được xem xét kỹ trong chương 2 của giáo trình này.

c. Kiểu mẫu tin (cấu trúc)

Nếu kiểu dữ liệu mảng là kiểu dữ liệu trong đó mỗi phần tử của nó là một tập hợp có thứ tự các giá trị có cùng cấu trúc được lưu trữ

liên tiếp nhau trong bộ nhớ thì mẩu tin là kiểu dữ liệu mà trong đó mỗi phần tử của nó là tập hợp các giá trị ***có thể khác cấu trúc***. Kiểu mẩu tin cho phép chúng ta mô tả các đối tượng có cấu trúc phức tạp.

Khai báo tổng quát của kiểu struct như sau:

```
typedef struct <tên kiểu struct>{
   <KDL> <tên trường>;
   <KDL> <tên trường>;
   ...
}[<Name>];
```

Ví dụ: Để mô tả các thông tin về một con người, ta có thể khai báo một kiểu dữ liệu như sau:

```
struct tagNguoi
{
   char   HoTen[35];
   int    NamSinh;
   char   NoiSinh[40];
   char   GioiTinh;   //0: Nữ, 1: Nam
   char   DiaChi[50];
   char   Ttgd;       //0:Không có gia đình, 1: Có gia đình
}Nguoi;
```

Kiểu mẩu tin bổ sung những thiếu sót của kiểu mảng, giúp ta có khả năng thể hiện các đối tượng đa dạng của thế giới thực vào trong máy tính một cách dễ dàng và chính xác hơn.

c. Kiểu union

Kiểu union là một dạng cấu trúc dữ liệu đặc biệt của ngôn ngữ C. Nó rất giống với kiểu struct. Chỉ khác một điều, trong kiểu union, các trường được phép dùng chung một vùng nhớ. Hay nói

cách khác, cùng một vùng nhớ ta có thể truy xuất dưới các dạng khác nhau.

Khai báo tổng quát của kiểu union như sau:

```
typedef union  <tên kiểu union>{
   <KDL> <tên trường>;
   <KDL> <tên trường>;
   ...
}[<Name>];
```

Ví dụ, ta có thể định nghĩa kiểu số sau:

```
typedef union  tagNumber{
   int i;
   long l;
}Number;
```

Việc truy xuất đến một trường trong union được thực hiện hoàn toàn giống như trong struct. Giả sử có biến n kiểu Number. Khi đó, n.i cho ta một số kiểu int còn n.l cho ta một số kiểu long, nhưng cả hai đều dùng chung một vùng nhớ. Vì vậy, khi ta gán

```
n.l = 0xfd03;
```

thì giá trị của n.i cũng bị thay đổi (n.i sẽ bằng 3);

Việc dùng kiểu union rất có lợi khi cần khai báo các CTDL mà nội dung của nó thay đổi tùy trạng thái. Ví dụ để mô tả các thông tin về một con người ta có thể khai báo một kiểu dữ liệu như sau:

```
struct tagNguoi
{
   char   HoTen[35];
```

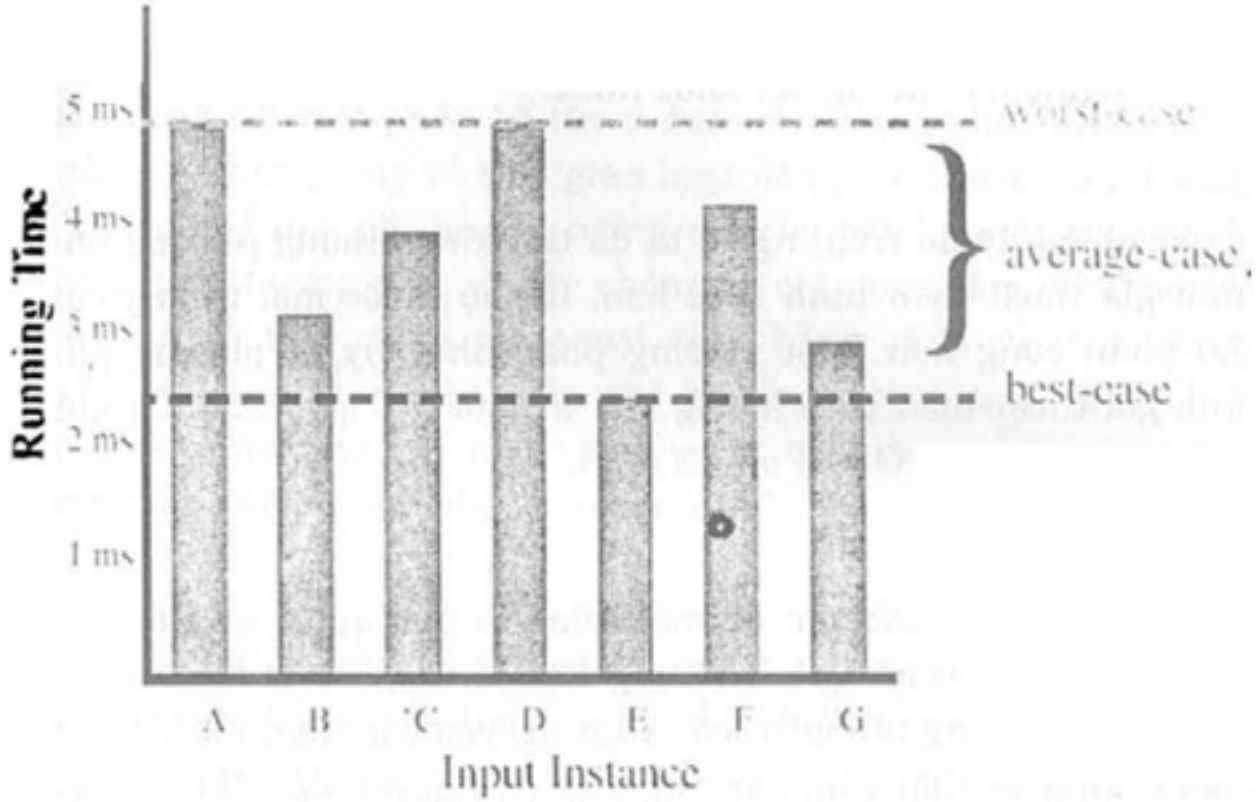
```
    int     NamSinh;
    char    NoiSinh[40];
    char    GioiTinh;    //0: Nữ, 1: Nam
    char    DiaChi[50];
    char    Ttgd;        //0:Không có gia đình, 1: Có gia đình
    union   {
        char tenVo[35];
        char tenChong[35];
    }
}Nguoi;
```

Tùy theo người mà ta đang xét là nam hay nữ ta sẽ truy xuất thông tin qua trường có tên tenVo hay tenChong.

IV. ĐÁNH GIÁ ĐỘ PHỨC TẠP GIẢI THUẬT

Hầu hết các bài toán đều có nhiều thuật toán khác nhau để giải quyết chúng. Như vậy, làm thế nào để chọn được sự cài đặt tốt nhất? Đây là một lĩnh vực được phát triển tốt trong nghiên cứu về khoa học máy tính. Chúng ta sẽ thường xuyên có cơ hội tiếp xúc với các kết quả nghiên cứu mô tả các tính năng của các thuật toán cơ bản. Tuy nhiên, việc so sánh các thuật toán rất cần thiết và chắc chắn rằng một vài dòng hướng dẫn tổng quát về phân tích thuật toán sẽ rất hữu dụng.

Khi nói đến hiệu quả của một thuật toán, người ta thường quan tâm đến chi phí cần dùng để thực hiện nó. Chi phí này thể hiện qua việc sử dụng tài nguyên như bộ nhớ, thời gian sử dụng CPU, ...Ta có thể đánh giá thuật toán bằng phương pháp thực nghiệm thông qua việc cài đặt thuật toán rồi chọn các bộ dữ liệu thử nghiệm. Thống kê các thông số nhận được khi chạy các dữ liệu này ta sẽ có một đánh giá về thuật toán.



Tuy nhiên, phương pháp thực nghiệm có một số nhược điểm sau khiến cho nó khó có khả năng áp dụng trên thực tế:

- Do phải cài đặt bằng một ngôn ngữ lập trình cụ thể nên thuật toán sẽ chịu sự hạn chế của ngữ lập trình này.

- Đồng thời, hiệu quả của thuật toán sẽ bị ảnh hưởng bởi trình độ của người cài đặt.

- Việc chọn được các bộ dữ liệu thử đặc trưng cho tất cả tập các dữ liệu vào của thuật toán là rất khó khăn và tốn nhiều chi phí.

- Các số liệu thu nhận được phụ thuộc nhiều vào phần cứng mà thuật toán được thử nghiệm trên đó. Điều này khiến cho

việc so sánh các thuật toán khó khăn nếu chúng được thử nghiệm ở những nơi khác nhau.

Vì những lý do trên, người ta đã tìm kiếm những phương pháp đánh giá thuật toán hình thức hơn, ít phụ thuộc môi trường cũng như phần cứng hơn. Một phương pháp như vậy là phương pháp đánh giá thuật toán theo hướng xấp xỉ tiệm cận qua các khái niệm toán học O-lớn O(), O-nhỏ o(), Ω(), Ξ().

Thông thường các vấn đề mà chúng ta giải quyết có một "kích thước" tự nhiên (thường là số lượng dữ liệu được xử lý) mà chúng ta sẽ gọi là N. Chúng ta muốn mô tả tài nguyên cần được dùng (thông thường nhất là thời gian cần thiết để giải quyết vấn đề) như một hàm số theo N. Chúng ta quan tâm đến **trường hợp trung bình**, tức là thời gian cần thiết để xử lý dữ liệu nhập thông thường, và cũng quan tâm đến **trường hợp xấu nhất**, tương ứng với thời gian cần thiết khi dữ liệu rơi vào trường hợp xấu nhất có thể có.

Việc xác định chi phí trong trường hợp trung bình thường được quan tâm nhiều nhất vì nó đại diện cho đa số trường hợp sử dụng thuật toán. Tuy nhiên, việc xác định chi phí trung bình này lại gặp nhiều khó khăn. Vì vậy, trong nhiều trường hợp, người ta xác định chi phí trong trường hợp xấu nhất (chặn trên) thay cho việc xác định chi phí trong trường hợp trung bình. Hơn nữa, trong một số bài toán, việc xác định chi phí trong trường hợp xấu nhất là rất quan trọng. Ví dụ, các bài toán trong hàng không, phẫu thuật, ...

1. Các bước phân tích thuật toán

Bước đầu tiên trong việc phân tích một thuật toán là xác định

đặc trưng dữ liệu sẽ được dùng làm dữ liệu nhập của thuật toán và quyết định phân tích nào là thích hợp. Về mặt lý tưởng, chúng ta muốn rằng với một phân bố tùy ý được cho của dữ liệu nhập, sẽ có sự phân bố tương ứng về thời gian hoạt động của thuật toán. Chúng ta không thể đạt tới điều lý tưởng này cho bất kỳ một thuật toán không tầm thường nào, vì vậy chúng ta chỉ quan tâm đến bao của thống kê về tính năng của thuật toán bằng cách cố gắng chứng minh thời gian chạy luôn luôn nhỏ hơn một "chặn trên" **bất chấp dữ liệu** nhập như thế nào và cố gắng tính được thời gian chạy trung bình cho dữ liệu nhập "ngẫu nhiên".

Bước thứ hai trong phân tích một thuật toán là nhận ra các thao tác trừu tượng của thuật toán để tách biệt sự phân tích với sự cài đặt. Ví dụ, chúng ta tách biệt sự nghiên cứu có bao nhiêu phép so sánh trong một thuật toán sắp xếp khỏi sự xác định cần bao nhiêu micro giây trên một máy tính cụ thể; yếu tố thứ nhất được xác định bởi tính chất của thuật toán, yếu tố thứ hai lại được xác định bởi tính chất của máy tính. Sự tách biệt này cho phép chúng ta so sánh các thuật toán một cách độc lập với sự cài đặt cụ thể hay độc lập với một máy tính cụ thể.

Bước thứ ba trong quá trình phân tích thuật toán là sự phân tích về mặt toán học, với mục đích tìm ra các giá trị trung bình và trường hợp xấu nhất cho mỗi đại lượng cơ bản. Chúng ta sẽ không gặp khó khăn khi tìm một chặn trên cho thời gian chạy chương trình, vấn đề ở chỗ là phải tìm ra chặn trên tốt nhất, tức là thời gian chạy chương trình khi gặp dữ liệu nhập của trường hợp xấu nhất. Trường hợp trung bình thông thường đòi hỏi một phân tích toán học tinh vi hơn trường hợp xấu nhất. Mỗi khi đã hoàn thành một quá trình phân tích thuật toán dựa vào các đại lượng cơ bản, nếu thời gian kết hợp với mỗi đại lượng được xác định rõ thì ta sẽ có các biểu thức để tính thời gian chạy.

Nói chung, tính năng của một thuật toán thường có thể được phân tích ở một mức độ vô cùng chính xác, chỉ bị giới hạn bởi tính năng không chắc chắn của máy tính hay bởi sự khó khăn trong việc xác định các tính chất toán học của một vài đại lượng trừu tượng. Tuy nhiên, thay vì phân tích một cách chi tiết chúng ta thường thích ước lượng để tránh sa vào chi tiết.

2. Sự phân lớp các thuật toán

Như đã được chú ý trong ở trên, hầu hết các thuật toán đều có một tham số chính là N, thông thường đó là số lượng các phần tử dữ liệu được xử lý mà ảnh hưởng rất nhiều tới thời gian chạy. Tham số N có thể là bậc của một đa thức, kích thước của một tập tin được sắp xếp hay tìm kiếm, số nút trong một đồ thị .v.v... Hầu hết tất cả các thuật toán trong giáo trình này có thời gian chạy tiệm cận tới một trong các hàm sau:

1. **Hằng số:** Hầu hết các chỉ thị của các chương trình đều được thực hiện một lần hay nhiều nhất chỉ một vài lần. Nếu tất cả các chỉ thị của cùng một chương trình có tính chất này thì chúng ta sẽ nói rằng thời gian chạy của nó là hằng số. Điều này hiển nhiên là hoàn cảnh phấn đấu để đạt được trong việc thiết kế thuật toán.

2. **logN**: Khi thời gian chạy của chương trình là **logarit** tức là thời gian chạy chương trình tiến chậm khi N lớn dần. Thời gian chạy thuộc loại này xuất hiện trong các chương trình mà giải một bài toán lớn bằng cách chuyển nó thành một bài toán nhỏ hơn, bằng cách cắt bỏ kích thước bớt một hằng số nào đó. Với mục đích của chúng ta, thời gian chạy có được xem như nhỏ hơn một hằng số "lớn". Cơ số của logarit làm thay đổi hằng số

đó nhưng không nhiều: khi N là một ngàn thì logN là 3 nếu cơ số là 10, là 10 nếu cơ số là 2; khi N là một triệu, logN được nhân gấp đôi. Bất cứ khi nào N được nhân đôi, logN tăng lên thêm một hằng số, nhưng logN không bị nhân gấp đôi khi N tăng tới N^2.

3. **N**: Khi thời gian chạy của một chương trình là **tuyến tính**, nói chung đây trường hợp mà một số lượng nhỏ các xử lý được làm cho mỗi phần tử dữ liệu nhập. Khi N là một triệu thì thời gian chạy cũng cỡ như vậy. Khi N được nhân gấp đôi thì thời gian chạy cũng được nhân gấp đôi. Đây là tình huống tối ưu cho một thuật toán mà phải xử lý N dữ liệu nhập (hay sản sinh ra N dữ liệu xuất).

4. **NlogN**: Đây là thời gian chạy tăng dần lên cho các thuật toán mà giải một bài toán bằng cách tách nó thành các bài toán con nhỏ hơn, kế đến giải quyết chúng một cách độc lập và sau đó tổ hợp các lời giải. Bởi vì thiếu một tính từ tốt hơn (có lẽ là "tuyến tính logarit"?), chúng ta nói rằng thời gian chạy của thuật toán như thế là "NlogN". Khi N là một triệu, NlogN có lẽ khoảng hai mươi triệu. Khi N được nhân gấp đôi, thời gian chạy bị nhân lên nhiều hơn gấp đôi (nhưng không nhiều lắm).

5. $\mathbf{N^2}$: Khi thời gian chạy của một thuật toán là **bậc hai**, trường hợp này chỉ có ý nghĩa thực tế cho các bài toán tương đối nhỏ. Thời gian bình phương thường tăng dần lên trong các thuật toán mà xử lý tất cả các cặp phần tử dữ liệu (có thể là hai vòng lặp lồng nhau). Khi N là một ngàn thì thời gian chạy là một triệu. Khi N được nhân đôi thì thời gian chạy tăng lên gấp bốn lần.

6. N^3:Tương tự, một thuật toán mà xử lý các bộ ba của các phần tử dữ liệu (có lẽ là ba vòng lặp lồng nhau) có thời gian chạy bậc ba và cũng chỉ có ý nghĩa thực tế trong các bài toán nhỏ. Khi N là một trăm thì thời gian chạy là một triệu. Khi N được nhân đôi, thời gian chạy tăng lên gấp tám lần.

7. 2^N: Một số ít thuật toán có thời gian chạy lũy thừa lại thích hợp trong một số trường hợp thực tế, mặc dù các thuật toán như thế là "sự ép buộc thô bạo" để giải các bài toán. Khi N là hai mươi thì thời gian chạy là một triệu. Khi N gấp đôi thì thời gian chạy được nâng lên lũy thừa hai!

Thời gian chạy của một chương trình cụ thể đôi khi là một hệ số hằng nhân với các số hạng nói trên ("số hạng dẫn đầu") cộng thêm một số hạng nhỏ hơn. Giá trị của hệ số hằng và các số hạng phụ thuộc vào kết quả của sự phân tích và các chi tiết cài đặt. Hệ số của số hạng dẫn đầu liên quan tới số chỉ thị bên trong vòng lặp: ở một tầng tùy ý của thiết kế thuật toán thì phải cẩn thận giới hạn số chỉ thị như thế. Với N lớn thì các số hạng dẫn đầu đóng vai trò chủ chốt; với N nhỏ thì các số hạng cùng đóng góp vào và sự so sánh các thuật toán sẽ khó khăn hơn. Trong hầu hết các trường hợp, chúng ta sẽ gặp các chương trình có thời gian chạy là "tuyến tính", "NlogN", "bậc ba", ... với hiểu ngầm là các phân tích hay nghiên cứu thực tế phải được làm trong trường hợp mà tính hiệu quả là rất quan trọng.

3. Phân tích trường hợp trung bình

Một tiếp cận trong việc nghiên cứu tính năng của thuật toán là khảo sát **trường hợp trung bình**. Trong tình huống đơn giản nhất, chúng ta có thể đặc trưng chính xác các dữ liệu nhập của thuật toán: ví dụ một thuật toán sắp xếp có thể thao tác trên một

mảng N số nguyên ngẫu nhiên, hay một thuật toán hình học có thể xử lý N điểm ngẫu nhiên trên mặt phẳng với các tọa độ nằm giữa 0 và 1. Kế đến là tính toán thời gian thực hiện trung bình của mỗi chỉ thị, và tính thời gian chạy trung bình của chương trình bằng cách nhân tần số sử dụng của mỗi chỉ thị với thời gian cần cho chỉ thị đó, sau cùng cộng tất cả chúng với nhau. Tuy nhiên có ít nhất ba khó khăn trong cách tiếp cận này như thảo luận dưới đây.

Trước tiên là trên một số máy tính rất khó xác định chính xác số lượng thời gian đòi hỏi cho mỗi chỉ thị. Trường hợp xấu nhất thì đại lượng này bị thay đổi và một số lượng lớn các phân tích chi tiết cho một máy tính có thể không thích hợp đối với một máy tính khác. Đây chính là vấn đề mà các nghiên cứu về độ phức tạp tính toán cũng cần phải né tránh.

Thứ hai, chính việc phân tích trường hợp trung bình lại thường là đòi hỏi toán học quá khó. Do tính chất tự nhiên của toán học thì việc chứng minh các chặn trên thì thường ít phức tạp hơn bởi vì không cần sự chính xác. Hiện nay chúng ta chưa biết được tính năng trong trường hợp trung bình của rất nhiều thuật toán.

Thứ ba (và chính là điều quan trọng nhất) trong việc phân tích trường hợp trung bình là mô hình dữ liệu nhập có thể không đặc trưng đầy đủ dữ liệu nhập mà chúng ta gặp trong thực tế. Ví dụ như làm thế nào để đặc trưng được dữ liệu nhập cho chương trình xử lý văn bản tiếng Anh? Một tác giả đề nghị nên dùng các mô hình dữ liệu nhập chẳng hạn như "tập tin thứ tự ngẫu nhiên" cho thuật toán sắp xếp, hay "tập hợp điểm ngẫu nhiên" cho thuật toán hình học, đối với những mô hình như thế thì có thể đạt được các kết quả toán học mà tiên đoán được tính năng của các chương trình chạy trên các các ứng dụng thông thường.

TÓM TẮT

Trong chương này, chúng ta đã xem xét các khái niệm về cấu trúc dữ liệu, kiểu dữ liệu. Thông thường, các ngôn ngữ lập trình luôn định nghĩa sẵn một số kiểu dữ liệu cơ bản. Các kiểu dữ liệu này thường có cấu trúc đơn giản. Để thể hiện được các đối tượng muôn hình vạn trạng trong thế giới thực, chỉ dùng các kiểu dữ liệu này là không đủ. Ta cần xây dựng các kiểu dữ liệu mới phù hợp với đối tượng mà nó biểu diễn. Thành phần dữ liệu luôn là một vấn đề quan trọng trong mọi chương trình. Vì vậy, việc thiết kế các cấu trúc dữ liệu tốt là một vấn đề đáng quan tâm.

Vấn đề thứ hai trong chương trình là các thuật toán (thuật giải). Một chương trình tốt phải có các cấu trúc dữ liệu phù hợp và các thuật toán hiệu quả. Khi khảo sát các thuật toán, chúng ta quan tâm đến chi phí thực hiện thuật toán. Chi phí này bao gồm chi phí về tài nguyên và thời gian cần để thực hiện thuật toán. Nếu như những đòi hỏi về tài nguyên có thể dễ dàng xác định thì việc xác định thời gian thực hiện nó không đơn giản. Có một số cách khác nhau để ước lượng khoảng thời gian này. Tuy nhiên, cách tiếp cận hợp lý nhất là hướng xấp xỉ tiệm cận. Hướng tiếp cận này không phụ thuộc ngôn ngữ, môi trường cài đặt cũng như trình độ của lập trình viên. Nó cho phép so sánh các thuật toán được khảo sát ở những nơi có vị trí địa lý rất xa nhau. Tuy nhiên, khi đánh giá ta cần chú ý thêm đến hệ số vô hướng trong kết quả đánh giá. Có khi hệ số này ảnh hưởng đáng kể đến chi phí thực của thuật toán.

Do việc đánh giá chi phí thực hiện trung bình của thuật toán thường phức tạp nên người ta thường đánh giá chi phí thực hiện thuật toán trong trường hợp xấu nhất. Hơn nữa, trong một số lớp thuật toán, việc xác định trường hợp xấu nhất là rất quan trọng.

BÀI TẬP

Bài tập lý thuyết

1. Tìm thêm một số ví dụ minh hoạ mối quan hệ giữa cấu trúc dữ liệu và giải thuật.

2. Cho biết một số kiểu dữ liệu được định nghĩa sẵn trong một ngôn ngữ lập trình các bạn thường sử dụng. Cho biết một số kiểu dữ liệu tiền định này có đủ để đáp ứng mọi yêu cầu về tổ chức dữ liệu không ?

3. Một ngôn ngữ lập trình có nên cho phép người sử dụng tự định nghĩa thêm các kiểu dữ liệu có cấu trúc ? Giải thích và cho ví dụ.

4. Cấu trúc dữ liệu và cấu trúc lưu trữ khác nhau những điểm nào ? Một cấu trúc dữ liệu có thể có nhiều cấu trúc lưu trữ được không ? Ngược lại, một cấu trúc lưu trữ có thể tương ứng với nhiều cấu trúc dữ liệu được không ? Cho ví dụ minh hoạ.

5. Giả sử có một bảng giờ tàu cho biết thông tin về các chuyến tàu khác nhau của mạng đường sắt. Hãy biểu diễn các dữ liệu này bằng một cấu trúc dữ liệu thích hợp (file, array, struct ...) sao cho dễ dàng truy xuất giờ khởi hành, giờ đến của một chuyến tàu bất kỳ tại một nhà ga bất kỳ.

Bài tập thực hành

6. Giả sử quy tắc tổ chức quản lý nhân viên của một công ty như sau :

- Thông tin về một nhân viên bao gồm lý lịch và bảng chấm công :

+ Lý lịch nhân viên :

- Mã nhân viên : chuỗi 8 ký tự
- Tên nhân viên : chuỗi 20 ký tự
- Tình trạng gia đình : 1 ký tự (M = Married, S = Single)
- Số con : số nguyên ≤ 20
- Trình độ văn hoá: chuỗi 2 ký tự
  (C1 = cấp 1 ; C2 = cấp 2 ; C3 = cấp 3 ;
  ĐH = đại học; CH = cao học)
- Lương căn bản : số ≤ 1000000

+ Chấm công nhân viên :

- Số ngày nghỉ có phép trong tháng : số ≤ 28
- Số ngày nghỉ không phép trong tháng : số ≤ 28
- Số ngày làm thêm trong tháng : số ≤ 28
- Kết qủa công việc : chuỗi 2 ký tự
  (T = Tốt; TB = Đạt ;K = Kém)
- Lương thực lĩnh trong tháng : số ≤ 2000000

- Quy tắc tính lương :

Lương thực lĩnh = Lương căn bản + Phụ trội
trong đó nếu:

- số con > 2: Phụ trội = +5% Lương căn bản
- trình độ văn hoá = CH: Phụ trội = +10%Lương căn bản
- làm thêm: Phụ trội=+4%Lương căn bản/ngày

- nghỉ không phép: Phụ trội= -5%Lương căn bản/ngày

- Chức năng yêu cầu :

  - Cập nhật lý lịch, bảng chấm công cho nhân viên (thêm, xoá, sửa)
  - Xem bảng lương hàng tháng
  - Tìm thông tin của một nhân viên

Tổ chức cấu trúc dữ liệu thích hợp để biểu diễn các thông tin trên, và cài đặt chương trình theo các chức năng đã mô tả.

Lưu ý

- Nên phân biệt các thông tin mang tính chất tĩnh (lý lịch) và động (chấm công hàng tháng)
- Số lượng nhân viên tối đa là 50 người

CHƯƠNG 2

TÌM KIẾM VÀ SẮP XẾP

Mục tiêu

- Giới thiệu một số thuật toán tìm kiếm và sắp xếp trong.
- Phân tích, đánh giá độ phức tạp của các giải thuật tìm kiếm, sắp xếp

I. NHU CẦU TÌM KIẾM VÀ SẮP XẾP DỮ LIỆU TRONG MỘT HỆ THỐNG THÔNG TIN

Trong hầu hết các hệ lưu trữ, quản lý dữ liệu, thao tác tìm kiếm thường được thực hiện nhất để khai thác thông tin :

Ví dụ: tra cứu từ điển, tìm sách trong thư viện...

Do các hệ thống thông tin thường phải lưu trữ một khối lượng dữ liệu đáng kể, nên việc xây dựng các giải thuật cho phép tìm kiếm nhanh sẽ có ý nghĩa rất lớn. Nếu dữ liệu trong hệ thống đã được tổ chức theo một trật tự nào đó, thì việc tìm kiếm sẽ tiến hành nhanh chóng và hiệu quả hơn:

Ví dụ: các từ trong từ điển được sắp xếp theo từng vần, trong mỗi vần lại được sắp xếp theo trình tự alphabet; sách trong thư viện được xếp theo chủ đề ...

Vì thế, khi xây dựng một hệ quản lý thông tin trên máy tính, bên cạnh các thuật toán tìm kiếm, các thuật toán sắp xếp dữ liệu cũng là một trong những chủ đề được quan tâm hàng đầu.

Hiện nay đã có nhiều giải thuật tìm kiếm và sắp xếp dược xây dựng, mức độ hiệu quả của từng giải thuật còn phụ thuộc vào tính chất của cấu trúc dữ liệu cụ thể mà nó tác động đến. Dữ liệu được lưu trữ chủ yếu trong bộ nhớ chính và trên bộ nhớ phụ, do đặc điểm khác nhau của thiết bị lưu trữ, các thuật toán tìm kiếm và sắp xếp được xây dựng cho các cấu trúc lưu trữ trên bộ nhớ chính hoặc phụ cũng có những đặc thù khác nhau. Chương này sẽ trình bày các thuật toán sắp xếp và tìm kiếm dữ liệu được lưu trữ trên bộ nhớ chính - gọi là các giải thuật ***tìm kiếm và sắp xếp nội***.

II. CÁC GIẢI THUẬT TÌM KIẾM NỘI

Có hai giải thuật thường được áp dụng để tìm kiếm dữ liệu là *tìm tuyến tính* và *tìm nhị phân*. Để đơn giản trong việc trình bày giải thuật, bài toán được đặc tả như sau:

- Tập dữ liệu được lưu trữ là dãy số $\mathbf{a_1}$, $\mathbf{a_2}$, ... ,$\mathbf{a_N}$. Giả sử chọn cấu trúc dữ liệu mảng để lưu trữ dãy số này trong bộ nhớ chính, có khai báo : `int a[N];`

  Lưu ý các bản cài đặt trong giáo trình sử dụng ngôn ngữ C, do đó chỉ số của mảng mặc định bắt đầu từ 0, nên các giá trị của các chỉ số có chênh lệch so với thuật toán, nhưng ý nghĩa không đổi.

- Khoá cần tìm là **x**, được khai báo như sau:

```
intx;
```

1. Tìm kiếm tuyến tính

- **Giải thuật**

Tìm tuyến tính là một kỹ thuật tìm kiếm rất đơn giản và cổ điển. Thuật toán tiến hành so sánh **x** lần lượt với phần tử thứ nhất, thứ hai, ... của mảng **a** cho đến khi gặp được phần tử có khóa cần tìm, hoặc đã tìm hết mảng mà không thấy **x**. Các bước tiến hành như sau :

Bước 1: i = 1; // bắt đầu từ phần tử đầu tiên của dãy

Bước 2: So sánh a[i] với x, có hai khả năng :
+ a[i] = x : Tìm thấy. Dừng
+ a[i] ≠ x : Sang Bước 3.

Bước 3: i = i + 1; // xét tiếp phần tử kế trong mảng
Nếu i > N: Hết mảng, không tìm thấy. Dừng
Ngược lại: Lặp lại Bước 2.

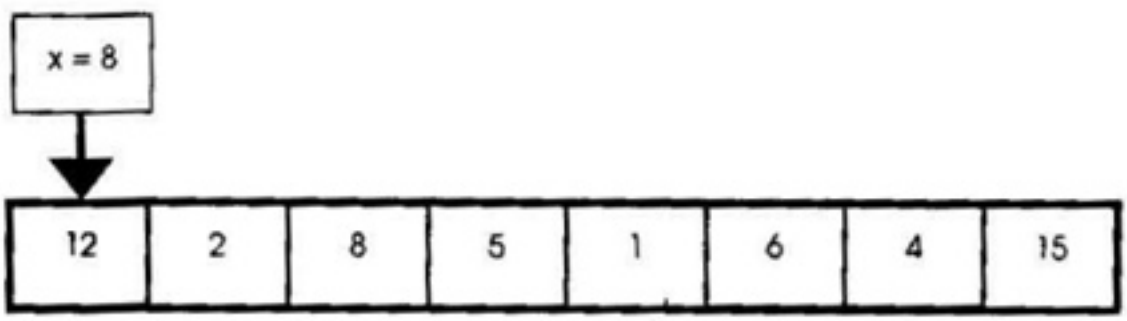
- **Ví dụ**

Cho dãy số a:

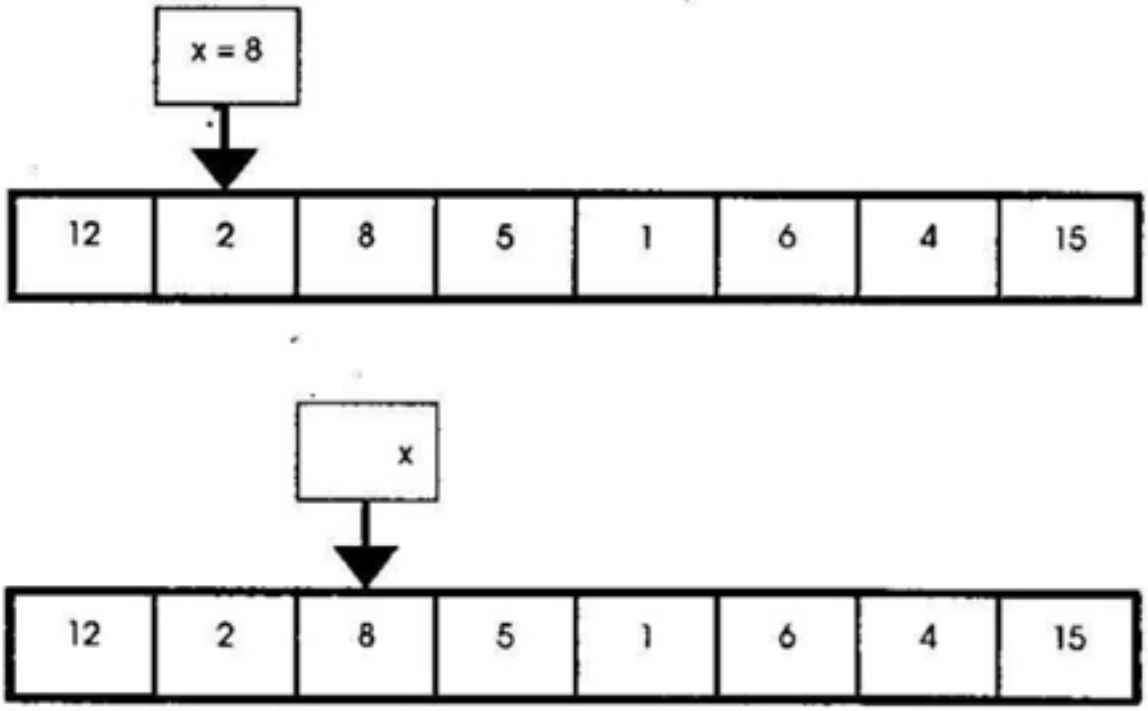
12 2 8 5 1 6 4 15

Nếu giá trị cần tìm là 8, giải thuật được tiến hành như sau :

i = 1



i = 2



i = 3

Dừng.

- **Cài đặt**

Từ mô tả trên đây của thuật toán tìm tuyến tính , có thể cài đặt hàm `LinearSearch` để xác định vị trí của phần tử có khoá x trong mảng a :

```
int LinearSearch(int a[], int N, int x)
{  int   i=0;
   while ((i<N) && (a[i]!=x ))     i++;
   if(i==N) return -1;   // tìm hết mảng nhưng không có x
      else  return i;    // a[i] là phần tử có khoá x
}
```

Trong cài đặt trên đây, nhận thấy mỗi lần lặp của vòng lặp `while` phải tiến thành kiểm tra hai điều kiện `(i<N)` - điều kiện biên của mảng - và `(a[i]!=x )`- điều kiện kiểm tra chính. Nhưng thật sự chỉ cần kiểm tra điều kiện chính `(a[i] !=x)`, để cải tiến cài đặt, có thể dùng phương pháp *"lính canh"* - đặt thêm một phần tử có giá trị x vào cuối mảng, như vậy bảo đảm luôn tìm thấy x trong mảng, sau đó dựa vào vị trí tìm thấy để kết luận. Cài đặt cải tiến sau đây của hàm `LinearSearch` giúp giảm bớt một phép so sánh trong vòng lặp :

```
int LinearSearch(int a[],int N,int x)
{  int   i=0;        // mảng gồm N phần tử từ a[0]..a[N-1]
   a[N] = x;     // thêm phần tử thứ N+1
   while (a[i]!=x )  i++;
   if (i==N)
      return -1;    // tìm hết mảng nhưng không có x
   else
      return i;     // tìm thấy x tại vị trí i
}
```

- **Đánh giá giải thuật**

Có thể ước lượng độ phức tạp của giải thuật tìm kiếm qua số lượng các phép so sánh được tiến hành để tìm ra x. Trường hợp giải thuật tìm tuyến tính, có:

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1	Phần tử đầu tiên có giá trị x
Xấu nhất	n+1	Phần tử cuối cùng có giá trị x
Trung bình	(n+1)/2	Giả sử xác suất các phần tử trong mảng nhận giá trị x là như nhau.

Vậy giải thuật tìm tuyến tính có độ phức tạp tính toán cấp n: T(n) = O(n)

NHẬN XÉT

- Giải thuật tìm tuyến tính không phụ thuộc vào thứ tự của các phần tử mảng, do vậy đây là phương pháp tổng quát nhất để tìm kiếm trên một dãy số bất kỳ.
- Một thuật toán có thể được cài đặt theo nhiều cách khác nhau, kỹ thuật cài đặt ảnh hưởng đến tốc độ thực hiện của thuật toán.

2. Tìm kiếm nhị phân

- **Giải thuật**

Đối với những dãy số đã có thứ tự (giả sử thứ tự tăng), các phần tử trong dãy có quan hệ $a_{i-1} \le a_i \le a_{i+1}$, từ đó kết luận được nếu $x > a_i$ thì x chỉ có thể xuất hiện trong đoạn $[a_{i+1}, a_N]$ của dãy , ngược lại nếu $x < a_i$ thì x chỉ có thể xuất hiện trong đoạn $[a_1, a_{i-1}]$ của dãy . Giải thuật tìm nhị phân áp dụng nhận xét trên đây để tìm cách giới hạn phạm vi tìm kiếm sau mỗi lần so sánh x với một phần tử trong dãy. Ý tưởng của giải thuật là tại mỗi bước tiến hành so sánh x với phần tử nằm ở vị trí giữa của dãy tìm kiếm hiện hành, dựa vào kết quả so sánh này để quyết định giới hạn dãy tìm kiếm ở bước kế tiếp là nửa trên hay nửa dưới của dãy tìm kiếm hiện hành. Giả sử dãy tìm kiếm hiện hành bao gồm các phần tử $a_{left} .. a_{right}$, các bước tiến hành như sau :

Bước 1: left = 1; right = N; // tìm kiếm trên tất cả các phần tử

Bước 2: mid = (left + right)/2; // lấy mốc so sánh
So sánh a[mid] với x, có ba khả năng :
+ a[mid] = x: Tìm thấy. Dừng

+ a[mid] > x: //tìm tiếp x trong dãy con $a_{left} .. a_{mid-1}$:
right = midle - 1;
+ a[mid] < x: //tìm tiếp x trong dãy con $a_{mid+1} .. a_{right}$:
left = mid + 1;

Bước 3: Nếu left ≤ right //còn phần tử chưa xét ⇒ tìm tiếp.
Lặp lại Bước 2.
Ngược lại: Dừng; //Đã xét hết tất cả các phần tử.

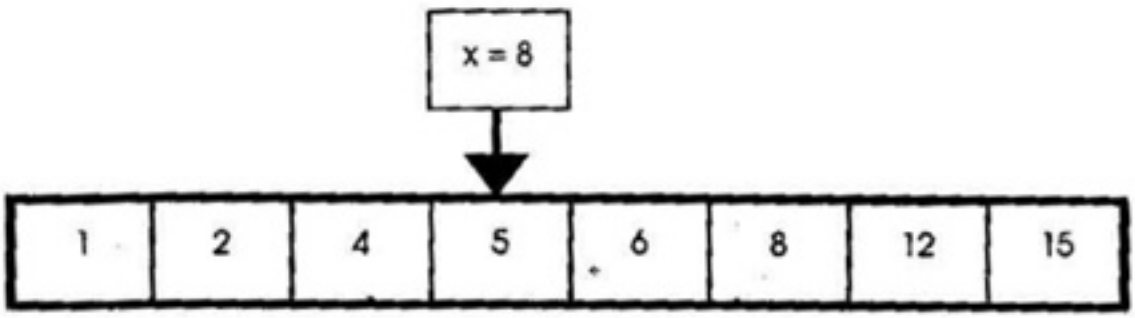
- **Ví dụ**

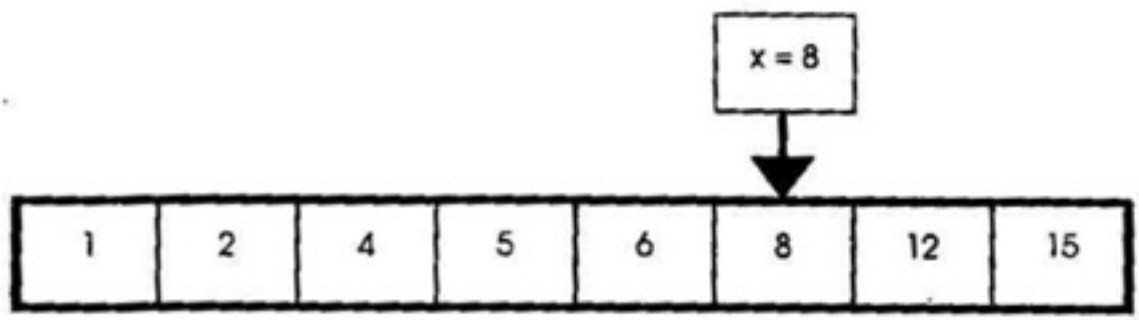
Cho dãy số a gồm 8 phần tử:

1 2 4 5 6 8 12 15

Nếu giá trị cần tìm là 8, giải thuật được tiến hành như sau:

left = 1, right = 8, midle = 4





left = 5, right = 8, midle = 6

Dừng.

- **Cài đặt**

Thuật toán tìm nhị phân có thể được cài đặt thành hàm BinarySearch:

```
int BinarySearch(int a[],int N,int x )
{  int   left =0; right = N-1;
   int   midle;
   do {
      mid = (left + right)/2;
      if (x = a[midle]) return midle;//Thấy x tại
mid
      else
         if (x < a[midle])    right = midle -1;
         else  left = midle +1;
   }while (left <= right);
   return -1; // Tìm hết dãy mà không có x
}
```

- **Đánh giá giải thuật**

Trường hợp giải thuật tìm nhị phân, có bảng phân tích sau :

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1	Phần tử giữa của mảng có giá trị x
Xấu nhất	$\log_2 n$	Không có x trong mảng
Trung bình	$\log_2 n/2$	Giả sử xác suất các phần tử trong mảng nhận giá trị x là như nhau

Vậy giải thuật tìm nhị phân có độ phức tạp tính toán cấp n: $T(n) = O(\log_2 n)$

NHẬN XÉT

- Giải thuật tìm nhị phân dựa vào quan hệ giá trị của các phần tử mảng để định hướng trong quá trình tìm kiếm, do vậy chỉ áp dụng được cho những dãy đã có thứ tự.
- Giải thuật tìm nhị phân tiết kiệm thời gian hơn rất nhiều so với giải thuật tìm tuyến tính do $T_{\text{nhị phân}}(n) = O(\log_2 n) < T_{\text{tuyến tính}}(n) = O(n)$.
- Tuy nhiên khi muốn áp dụng giải thuật tìm nhị phân

cần phải xét đến thời gian sắp xếp dãy số để thỏa điều kiện dãy số có thứ tự. Thời gian này không nhỏ, và khi dãy số biến động cần phải tiến hành sắp xếp lại ... Tất cả các nhu cầu đó tạo ra khuyết điểm chính cho giải thuật tìm nhị phân. Ta cần cân nhắc nhu cầu thực tế để chọn một trong hai giải thuật tìm kiếm trên sao cho có lợi nhất.

III. CÁC GIẢI THUẬT SẮP XẾP NỘI

1. Định nghĩa bài toán sắp xếp

Sắp xếp là quá trình xử lý một danh sách các phần tử (hoặc các mẩu tin) để đặt chúng theo một thứ tự thỏa mãn một tiêu chuẩn nào đó dựa trên nội dung thông tin lưu giữ tại mỗi phần tử.

Tại sao cần phải sắp xếp các phần tử thay, vì để nó ở dạng tự nhiên (chưa có thứ tự) vốn có ? Ví dụ của bài toán tìm kiếm với phương pháp tìm kiếm nhị phân và tuần tự đủ để trả lời câu hỏi này.

Khi khảo sát bài toán sắp xếp, ta sẽ phải làm việc nhiều với một khái niệm gọi là ***nghịch thế***.

Khái niệm nghịch thế

Xét một mảng các số $a_0, a_1, \ldots a_n$.
Nếu có $i<j$ và $a_i > a_j$, thì ta gọi đó là một nghịch thế.

Mảng chưa sắp xếp sẽ có nghịch thế.

Mảng đã có thứ tự sẽ không chứa nghịch thế. Khi đó a_0 sẽ là phần tử nhỏ nhất rồi đến a_1, a_2, ...

$a_0 \le a_1 \le ... \le a_n$

Như vậy, để sắp xếp một mảng, ta có thể tìm cách giảm số các nghịch thế trong mảng này bằng cách hoán vị các cặp phần tử a_i, a_j nếu có $i < j$ và $a_i > a_j$ theo một qui luật nào đó.

Cho trước một dãy số $a_1, a_2, ..., a_N$ được lưu trữ trong cấu trúc dữ liệu mảng

```
int   a[N];
```

Sắp xếp dãy số $a_1, a_2, ..., a_N$ là thực hiện việc bố trí lại các phần tử sao cho hình thành được dãy mới $a_{k1}, a_{k2}, ..., a_{kN}$ có thứ tự (giả sử xét thứ tự tăng) nghĩa là $a_{ki} \le a_{ki-1}$. Mà để quyết định được những tình huống cần thay đổi vị trí các phần tử trong dãy, cần dựa vào kết quả của một loạt phép so sánh. Chính vì vậy, hai thao tác so sánh và gán là các thao tác cơ bản của hầu hết các thuật toán sắp xếp.

Khi xây dựng một thuật toán sắp xếp cần chú ý tìm cách giảm thiểu những phép so sánh và đổi chỗ không cần thiết để tăng hiệu quả của thuật toán. Đối với các dãy số được lưu trữ trong bộ nhớ chính, nhu cầu tiết kiệm bộ nhớ được đặt nặng, do vậy những thuật toán sắp xếp đòi hỏi cấp phát thêm vùng nhớ để lưu trữ dãy kết quả ngoài vùng nhớ lưu trữ dãy số ban đầu thường ít được quan tâm. Thay vào đó, các thuật toán sắp xếp trực tiếp trên dãy số ban đầu - gọi là các thuật toán sắp xếp tại chỗ - lại được đầu tư phát triển. Phần này giới thiệu một số giải thuật sắp xếp từ đơn giản đến phức tạp có thể áp dụng thích hợp cho việc sắp xếp nội.

2. Các phương pháp sắp xếp thông dụng

Sau đây là một số phương pháp sắp xếp thông dụng sẽ được đề cập đến trong giáo trình này:

- Chọn trực tiếp - Selection sort
- Chèn trực tiếp - Insertion sort
- Đổi chỗ trực tiếp - Interchange sort
- Nổi bọt – Bubble sort
- Shaker sort
- Chèn nhị phân - Binary Insertion sort
- Shell sort
- Heap sort
- Quick sort
- Merge sort
- Radix sort

Chúng ta sẽ lần lượt khảo sát các thuật toán trên. Các thuật toán như Interchange sort, Bubble sort, Shaker sort, Insertion sort, Selection sort là những thuật toán đơn giản dễ cài đặt nhưng chi phí cao. Các thuật toán Shell sort, Heap sort, Quick sort, Merge sort phức tạp hơn nhưng hiệu suất cao hơn nhóm các thuật toán đầu. Cả hai nhóm thuật toán trên đều có một điểm chung là đều được xây dựng dựa trên cơ sở việc so sánh giá trị của các phần tử trong mảng (hay so sánh các khóa tìm kiếm). Riêng phương pháp Radix sort đại diện cho một lớp các thuật toán sắp xếp khác hẳn các thuật toán trước. Lớp thuật toán này không dựa trên giá trị của các phần tử để sắp xếp.

3. Phương pháp chọn trực tiếp

- **Giải thuật**

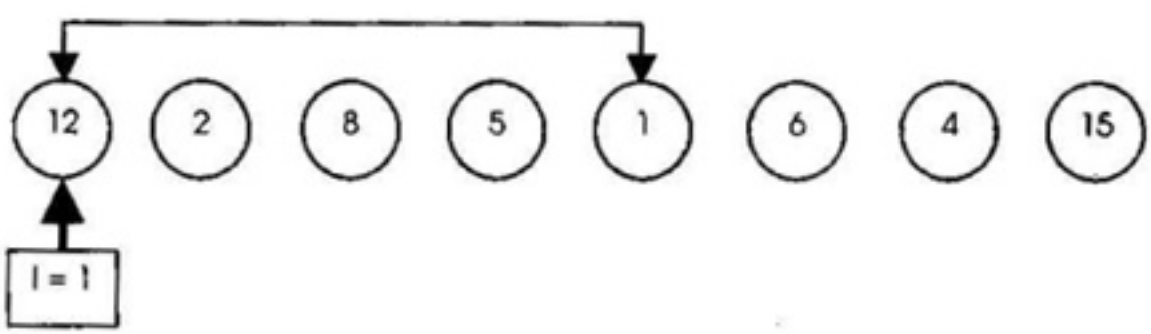
Ta thấy rằng, nếu mảng có thứ tự, phần tử a_i luôn là $\min(a_i, a_{i+1}, ..., a_{n-1})$. Ý tưởng của thuật toán chọn trực tiếp mô phỏng một trong những cách sắp xếp tự nhiên nhất trong thực tế: chọn phần tử nhỏ nhất trong N phần tử ban đầu, đưa phần tử này về vị trí đúng là đầu dãy hiện hành; sau đó không quan tâm đến nó nữa, xem dãy hiện hành chỉ còn N-1 phần tử của dãy ban đầu, bắt đầu từ vị trí thứ 2; lặp lại quá trình trên cho dãy hiện hành... đến khi dãy hiện hành chỉ còn 1 phần tử. Dãy ban đầu có N phần tử, vậy tóm tắt ý tưởng thuật toán là thực hiện N-1 lượt việc đưa phần tử nhỏ nhất trong dãy hiện hành về vị trí đúng ở đầu dãy. Các bước tiến hành như sau :

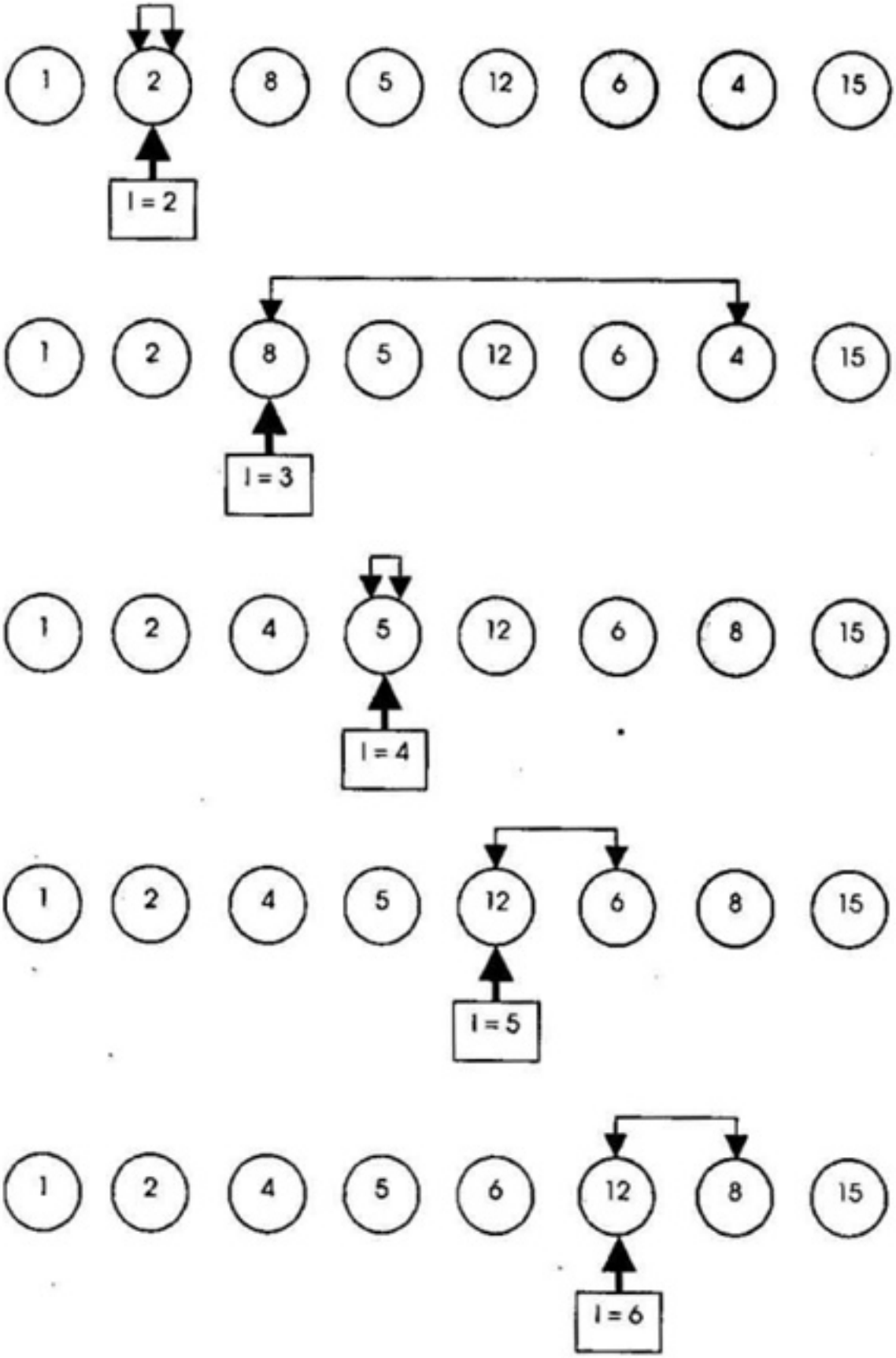
Bước 1 : i = 1;
Bước 2 : Tìm phần tử a[min] nhỏ nhất trong dãy hiện hành từ a[i] đến a[N]
Bước 3 : Hoán vị a[min] và a[i]
Bước 4 : Nếu i ≤ N-1 thì i = i+1; Lặp lại Bước 2
Ngược lại: Dừng. //N-1 phần tử đã nằm đúng vị trí.

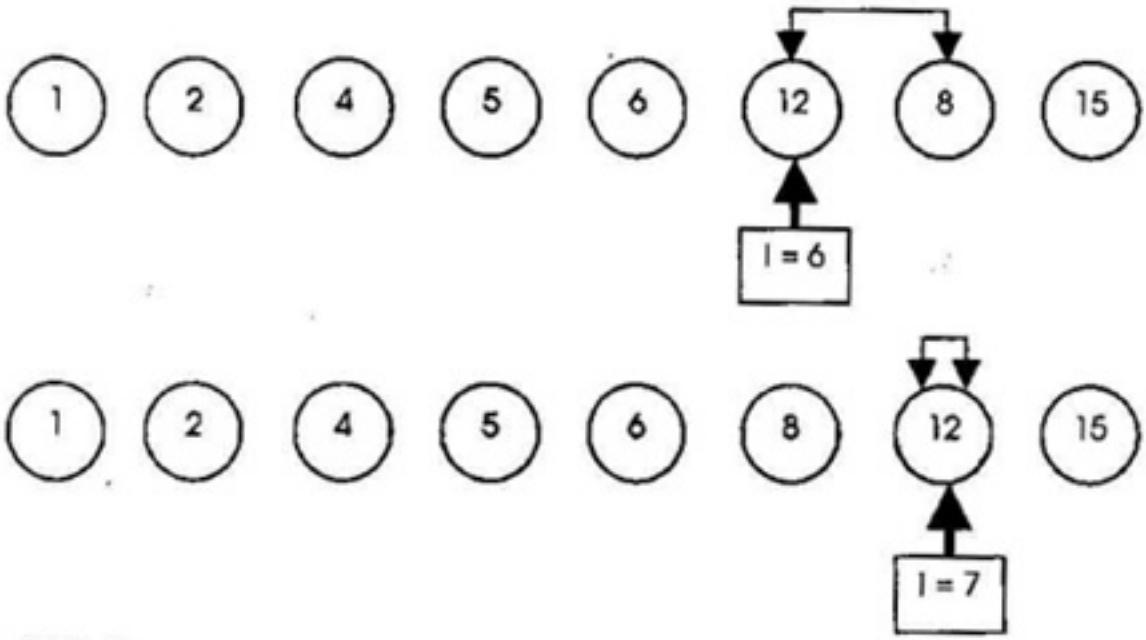
- **Ví dụ**

Cho dãy số a:

12 2 8 5 1 6 4 15




1
2
8
5
12
6
4
15
I = 2
1
2
8
5
12
6
4
15
I = 3
1
2
4
5
12
6
8
15
I = 4
1
2
4
5
12
6
8
15
I = 5
1
2
4
5
6
12
8
15
I = 6



- **Cài đặt**

Cài đặt thuật toán sắp xếp chọn trực tiếp thành hàm SelectionSort

```
void SelectionSort(int a[],int N )
{   int    min; // chỉ số phần tử nhỏ nhất trong dãy hiện hành
    for (int  i=0; i<N-1 ; i++)
    {
       min = i;
       for(int j = i+1; j <N ; j++)
          if (a[j ] < a[min])
               min = j; // ghi nhận vị trí phần tử hiện nhỏ
nhất
       Hoanvi(a[min], a[i]);
    }
}
```

- **Đánh giá giải thuật**

Đối với giải thuật chọn trực tiếp, có thể thấy rằng ở lượt thứ i, bao giờ cũng cần (n-i) lần so sánh để xác định phần tử nhỏ nhất hiện hành. Số lượng phép so sánh này không phụ thuộc vào tình trạng của dãy số ban đầu, do vậy trong mọi trường hợp có thể kết luận :

$$\text{Số lần so sánh} = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$$

Số lần hoán vị (một hoán vị bằng ba phép gán) lại phụ thuộc vào tình trạng ban đầu của dãy số, ta chỉ có thể ước lượng trong từng trường hợp như sau :

Trường hợp	Số lần so sánh	Số phép gán
Tốt nhất	n(n-1)/2	0
Xấu nhất	n(n-1)/2	3n(n-1)/2

4. Phương pháp chèn trực tiếp

- **Giải thuật**

Giả sử có một dãy a_1 , a_2 ,... ,a_n trong đó i phần tử đầu tiên a_1 , a_2 ,... ,a_{i-1} đã có thứ tự. Ý tưởng chính của giải thuật sắp xếp bằng phương pháp chèn trực tiếp là tìm cách chèn phần tử a_i vào vị trí thíchA hợp của đoạn đã được sắp để có dãy mới a_1 , a_2 ,... ,a_i trở nên có thứ tự. Vị trí này chính là vị trí giữa hai phần tử a_{k-1} và a_k thỏa $a_{k-1} \le a_i < a_k$ $(1 \le k \le i)$.

Cho dãy ban đầu a_1 , a_2 ,... ,a_n, ta có thể xem như đã có đoạn gồm một phần tử a_1 đã được sắp, sau đó thêm a_2 vào đoạn a_1 sẽ có đoạn a_1 a_2 được sắp; tiếp tục thêm a_3 vào đoạn a_1 a_2 để có đoạn a_1 a_2 a_3 được sắp; tiếp tục cho đến khi thêm xong a_N vào đoạn a_1 a_2 ...a_{N-1} sẽ có dãy a_1 a_2.... a_N được sắp. Các bước tiến hành như sau :

Bước 1: i = 2; // giả sử có đoạn a[1] đã được sắp

Bước 2: x = a[i]; Tìm vị trí pos thích hợp trong đoạn a[1]

đến a[i-1] để chèn a[i] vào

Bước 3: Dời chỗ các phần tử từ a[pos] đến a[i-1] sang phải 1 vị trí để dành chỗ cho a[i]

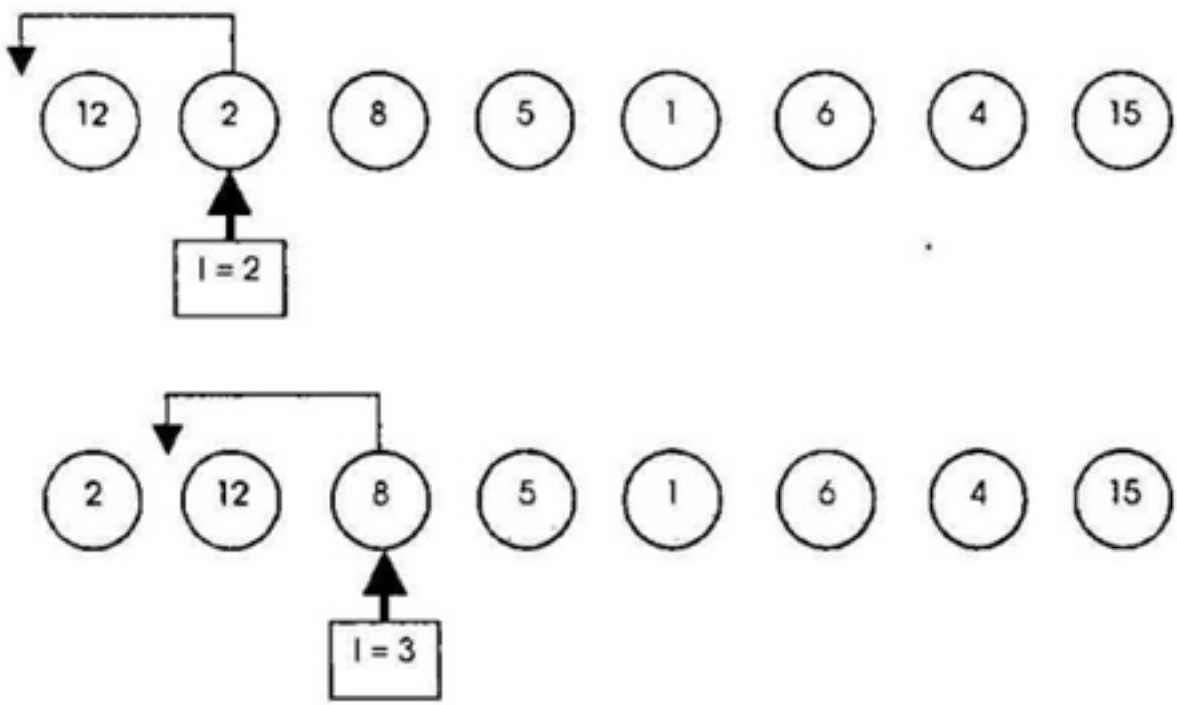
Bước 4: a[pos] = x; // có đoạn a[1]..a[i] đã được sắp

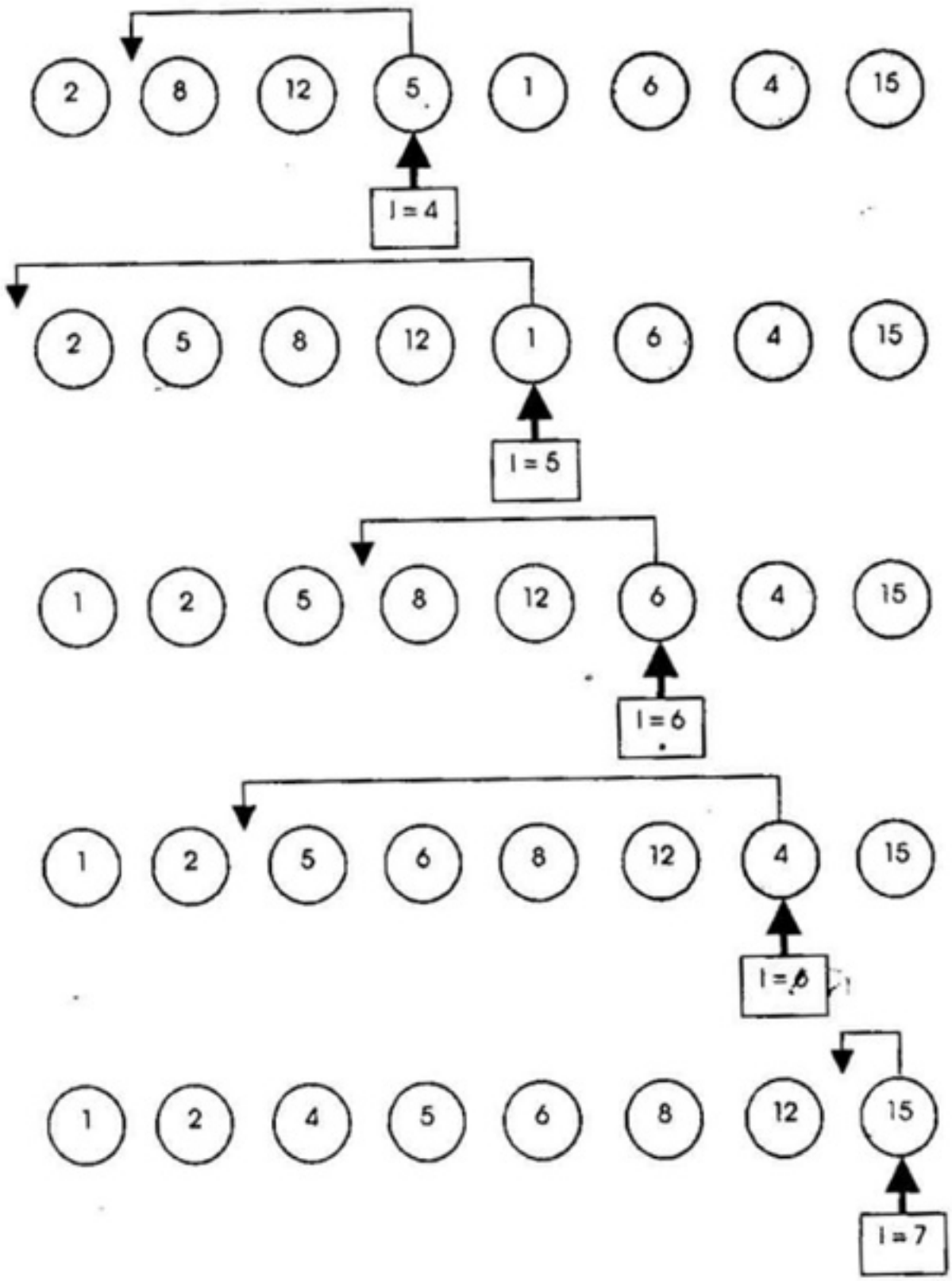
Bước 5: i = i+1;
Nếu i ≤ n : Lặp lại Bước 2.
Ngược lại : Dừng.

- **Ví dụ**

Cho dãy số a:

12 2 8 5 1 6 4 15




2 8 12 5 1 6 4 15
I = 4
2 5 8 12 1 6 4 15
I = 5
1 2 5 8 12 6 4 15
I = 6
1 2 5 6 8 12 4 15
I = 7
1 2 4 5 6 8 12 15

Dừng

- **Cài đặt**

Cài đặt thuật toán sắp xếp chèn trực tiếp thành hàm InsertionSort

```
void InsertionSort(int a[], int N )
{  int pos, i;
   int x; //lưu giá trị a[i] tránh bị ghi đè khi dời chỗ các phần tử.
   for(int i=1 ; i<N ; i++) //đoạn a[0] đã sắp
   {
      x = a[i]; pos = i-1;
      // tìm vị trí chèn x
      while((pos >= 0)&&(a[pos] > x))
      {// kết hợp dời chỗ các phần tử sẽ đứng sau x trong dãy mới
         a[pos+1] = a[pos];
         pos--;
      }
      a[pos+1] = x];// chèn x vào dãy
   }
}
```

NHẬN XÉT

Khi tìm vị trí thích hợp để chèn a[i] vào đoạn a[0] đến a[i-1], do đoạn đã được sắp, nên có thể sử dụng giải thuật tìm nhị phân để thực hiện việc tìm vị trí pos, khi đó có giải thuật sắp xếp chèn nhị phân :

```
void  BInsertionSort(int a[], int N )
{  int l,r,m,i;
   int x; //lưu giá trị a[i] tránh bị ghi đè khi dời chỗ các phần tử.
   for(int i=1 ; i<N ; i++)
   {  x = a[i]; l = 1; r = i-1;
      while(i<=r) // tìm vị trí chèn x
      {  m = (l+r)/2;     // tìm vị trí thích hợp m
         if(x < a[m])     r = m-1;
         else  l = m+1;
      }
```

```
            for(int j = i-1 ; j >=1 ; j--)
               a[j+1] = a[j];// dời các phần tử sẽ đứng sau x
            a[1] = x;                  // chèn x vào dãy
        }
    }
```

- **Đánh giá giải thuật**

Đối với giải thuật chèn trực tiếp, các phép so sánh xảy ra trong mỗi vòng lặp **while** tìm vị trí thích hợp **pos**, và mỗi lần xác định vị trí đang xét không thích hợp, sẽ dời chỗ phần tử a[pos] tương ứng. Giải thuật thực hiện tất cả N-1 vòng lặp **while** , do số lượng phép so sánh và dời chỗ này phụ thuộc vào tình trạng của dãy số ban đầu, nên chỉ có thể ước lược trong từng trường hợp như sau :

Trường hợp	Số phép so sánh	Số phép gán
Tốt nhất	$\sum_{i=1}^{n-1} 1 = n-1$	$\sum_{i=1}^{n-1} 2 = 2(n-1)$
Xấu nhất	$\sum_{i=1}^{n-1} (i-1) = \frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1} (i+1) = \frac{n(n+1)}{2} - 1$

5. Phương pháp đổi chỗ trực tiếp

- **Giải thuật**

Như đã đề cập ở đầu phần này, để sắp xếp một dãy số, ta có thể xét các nghịch thế có trong dãy và làm triệt tiêu dần chúng đi. Ý tưởng chính của giải thuật là xuất phát từ đầu dãy, tìm tất cả nghịch thế chứa phần tử này, triệt tiêu chúng bằng cách đổi chỗ phần tử này với phần tử tương ứng trong cặp nghịch thế. Lặp lại xử lý trên với các phần tử tiếp

theo trong dãy. Các bước tiến hành như sau :

Bước 1 : i = 1;// bắt đầu từ đầu dãy

Bước 2 : j = i+1;//tìm các phần tử a[j] < a[i], j>i

Bước 3 :

Trong khi j ≤ N thực hiện

Nếu a[j]<a[i]: a[i]↔a[j];//xét cặp phần tử a[i], a[j]

j = j+1;

Bước 4 : i = i+1;

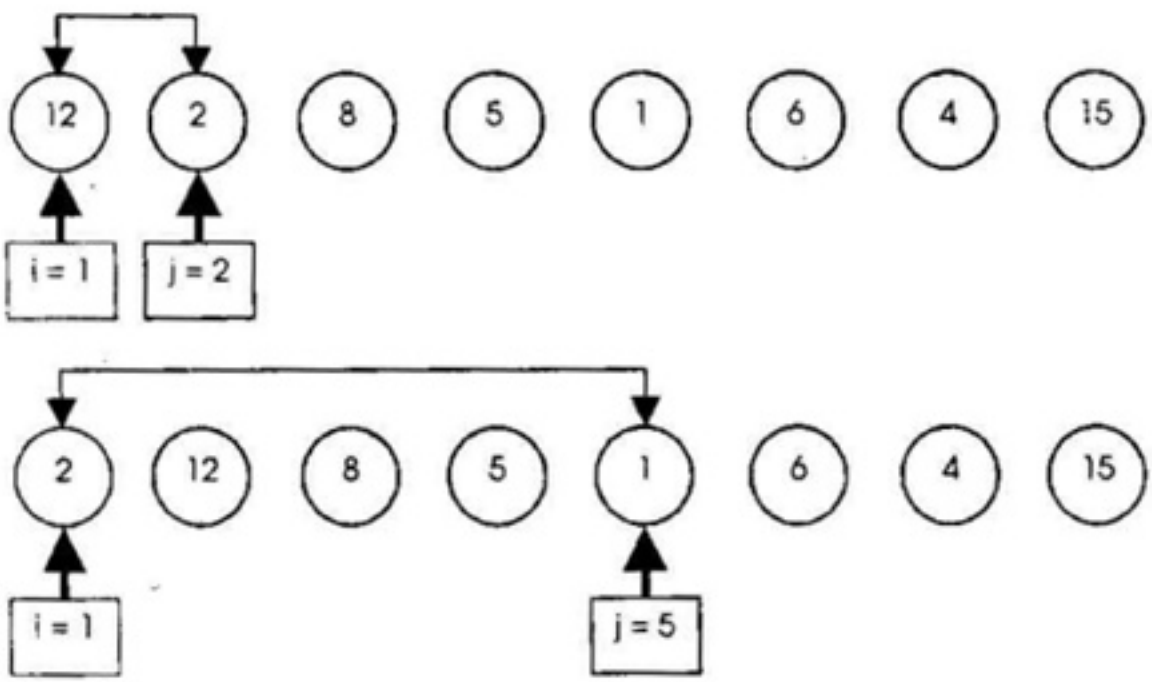
Nếu i < n: Lặp lại Bước 2.

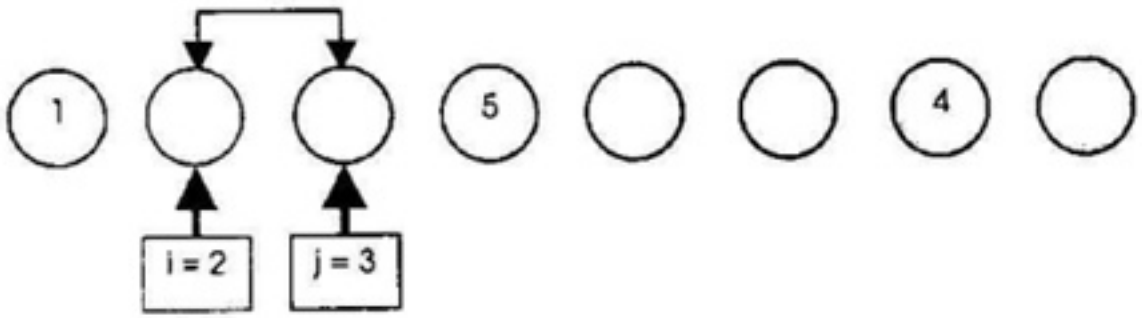
Ngược lại: Dừng.

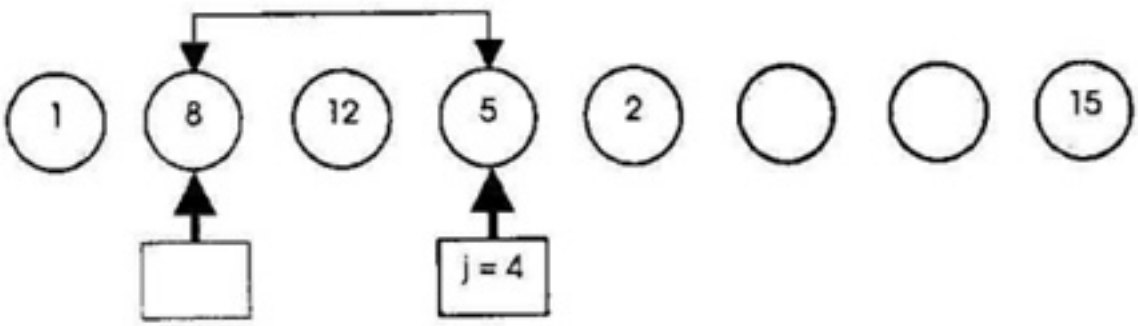
- **Ví dụ**

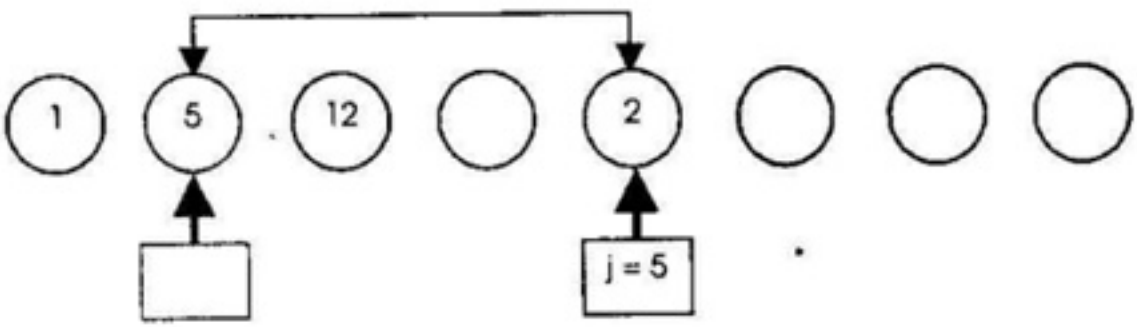
Cho dãy số a:

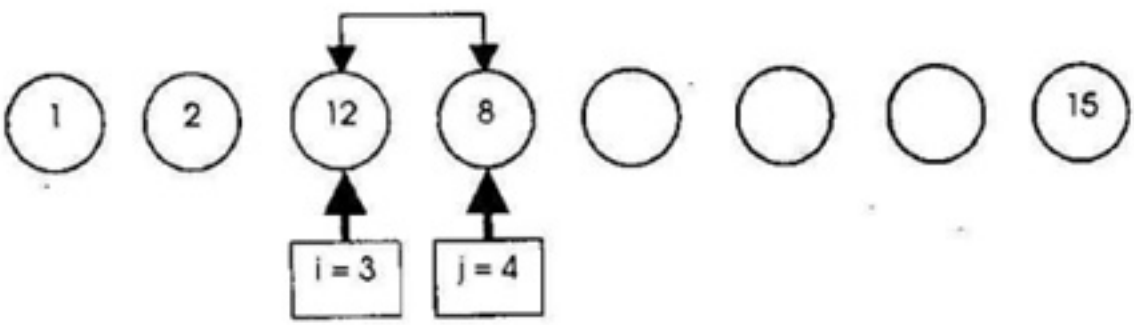
12 2 8 5 1 6 4 15

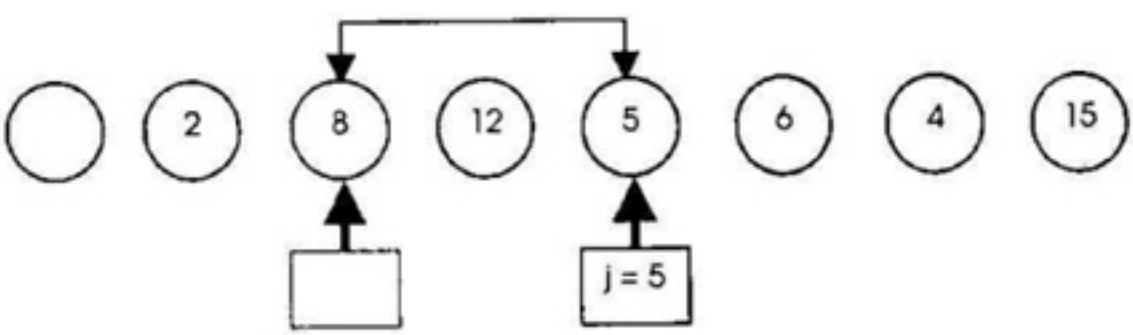


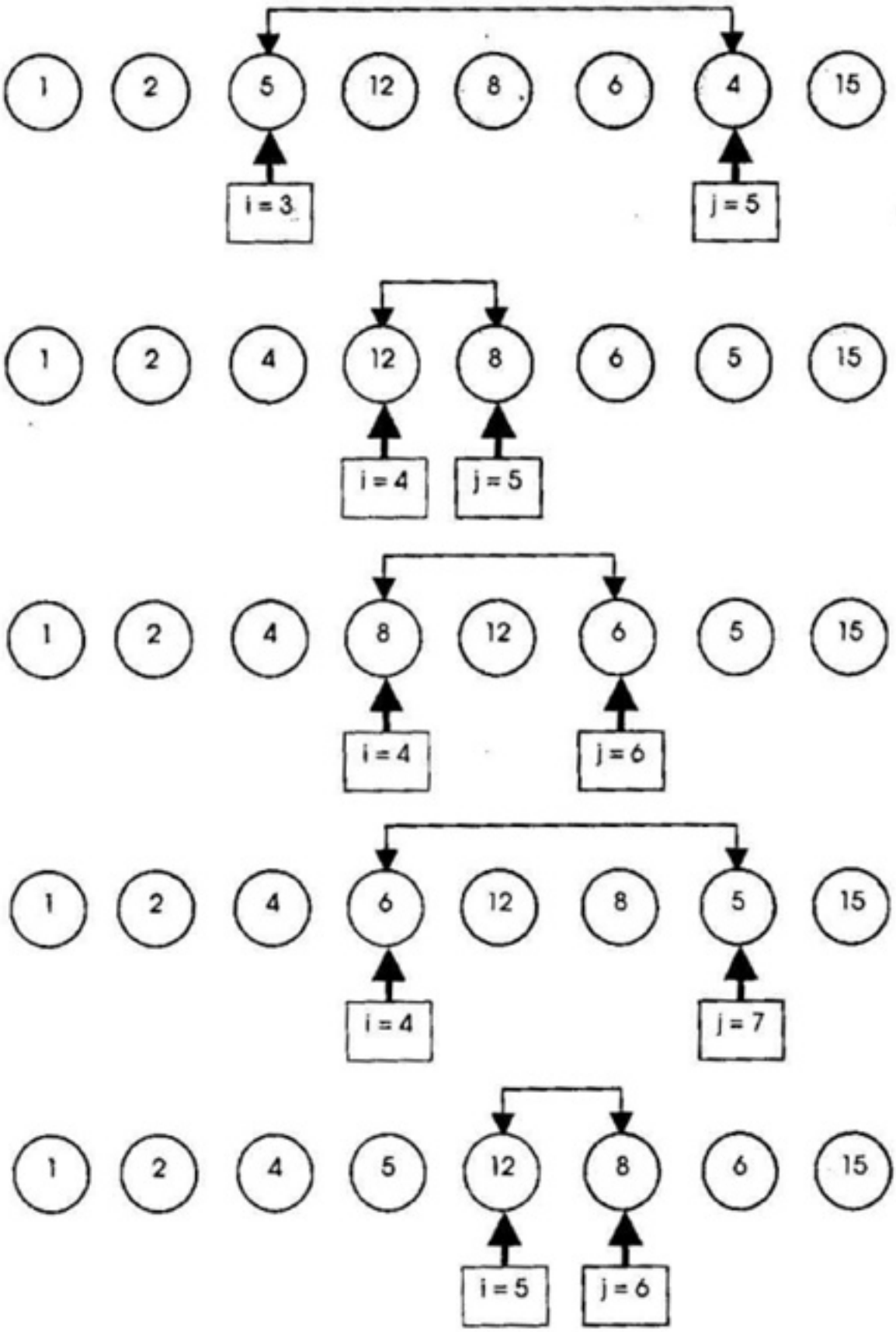

1
5
4
i = 2
j = 3

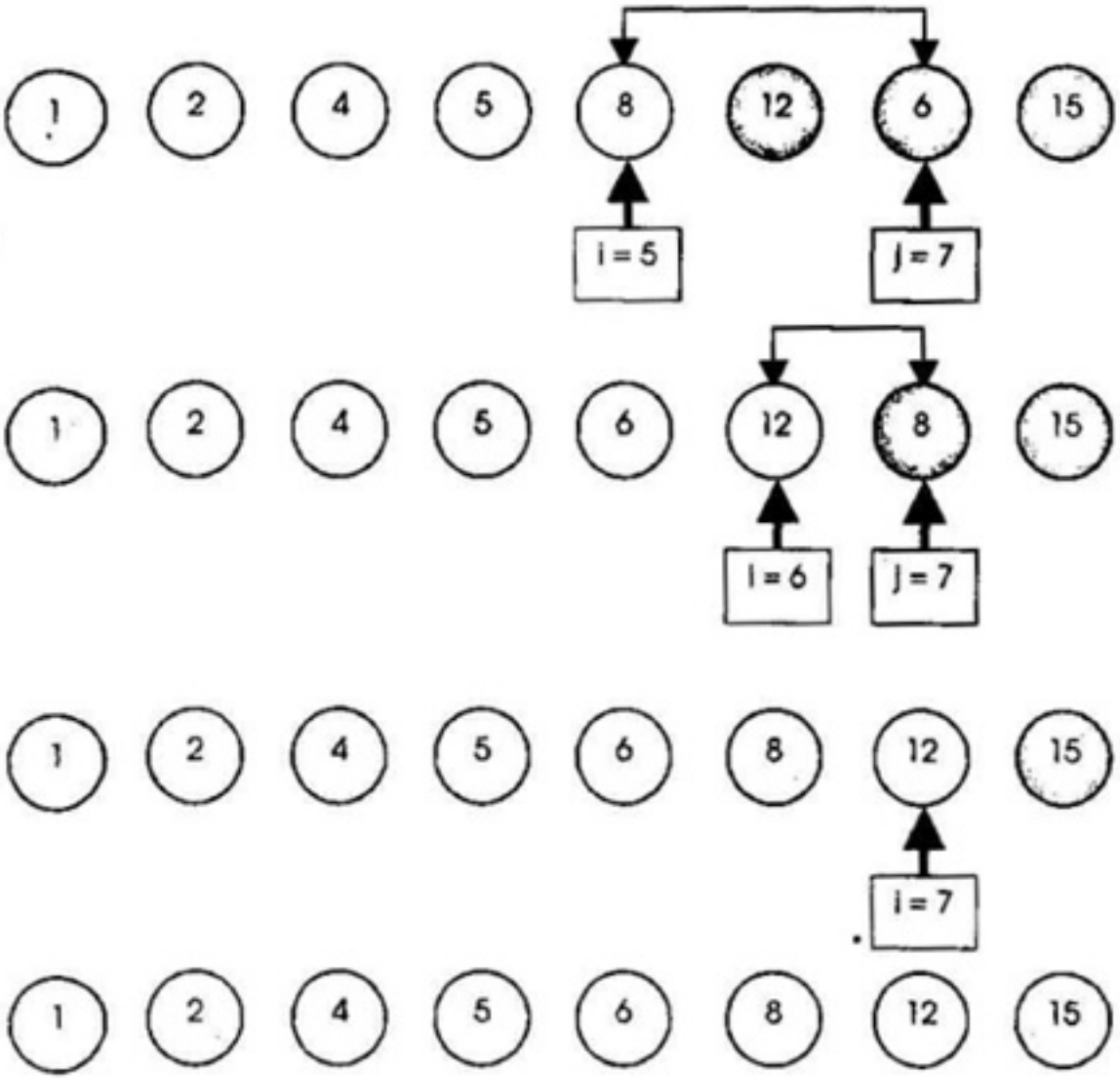

1
8
12
5
2
15
j = 4


1
5
12
2
j = 5


1
2
12
8
15
i = 3
j = 4


2
8
12
5
6
4
15
j = 5


1 2 5 12 8 6 4 15
i = 3
j = 5
1 2 4 12 8 6 5 15
i = 4
j = 5
1 2 4 8 12 6 5 15
i = 4
j = 6
1 2 4 6 12 8 5 15
i = 4
j = 7
1 2 4 5 12 8 6 15
i = 5
j = 6



- **Cài đặt**

Cài đặt thuật toán sắp xếp theo kiểu đổi chỗ trực tiếp thành hàm InterchangeSort:

```
void InterchangeSort(int a[], int N )
{  int   i, j;
   for (i = 0 ; i<N-1 ; i++)
      for (j =i+1; j < N ; j++)
      if(a[j ]< a[i])     // nếu có sự sai vị trí thì đổi chỗ
            Hoanvi(a[i],a[j]);
}
```

- **Đánh giá giải thuật**

Đối với giải thuật đổi chỗ trực tiếp, số lượng các phép so sánh xảy ra không phụ thuộc vào tình trạng của dãy số ban đầu, nhưng số lượng phép hoán vị thực hiện tùy thuộc vào kết quả so sánh, có thể ước lược trong từng trường hợp như sau :

Trường hợp	Số lần so sánh	Số lần hoán vị
Tốt nhất	$\sum_{i=1}^{n-1}(n-i+1)=\frac{n(n-1)}{2}$	0
Xấu nhất	$\frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1}(n-i+1)=\frac{n(n-1)}{2}$

6. Phương pháp nổi bọt (Bubble sort)

- **Giải thuật**

Ý tưởng chính của giải thuật là xuất phát từ cuối (đầu) dãy, đổi chỗ các cặp phần tử kế cận để đưa phần tử nhỏ (lớn) hơn trong cặp phần tử đó về vị trí đúng đầu(cuối) dãy hiện hành, sau đó sẽ không xét đến nó ở bước tiếp theo, do vậy ở lần xử lý thứ i sẽ có vị trí đầu dãy là i. Lặp lại xử lý trên cho đến khi không còn cặp phần tử nào để xét. Các bước tiến hành như sau

Bước 1 : i = 1; // lần xử lý đầu tiên

Bước 2 : j = N; //Duyệt từ cuối dãy ngược về vị trí i
Trong khi (j < i) thực hiện:
Nếu a[j]<a[j-1]: a[j]↔a[j-1];//xét cặp phần tử kế cận
j = j-1;

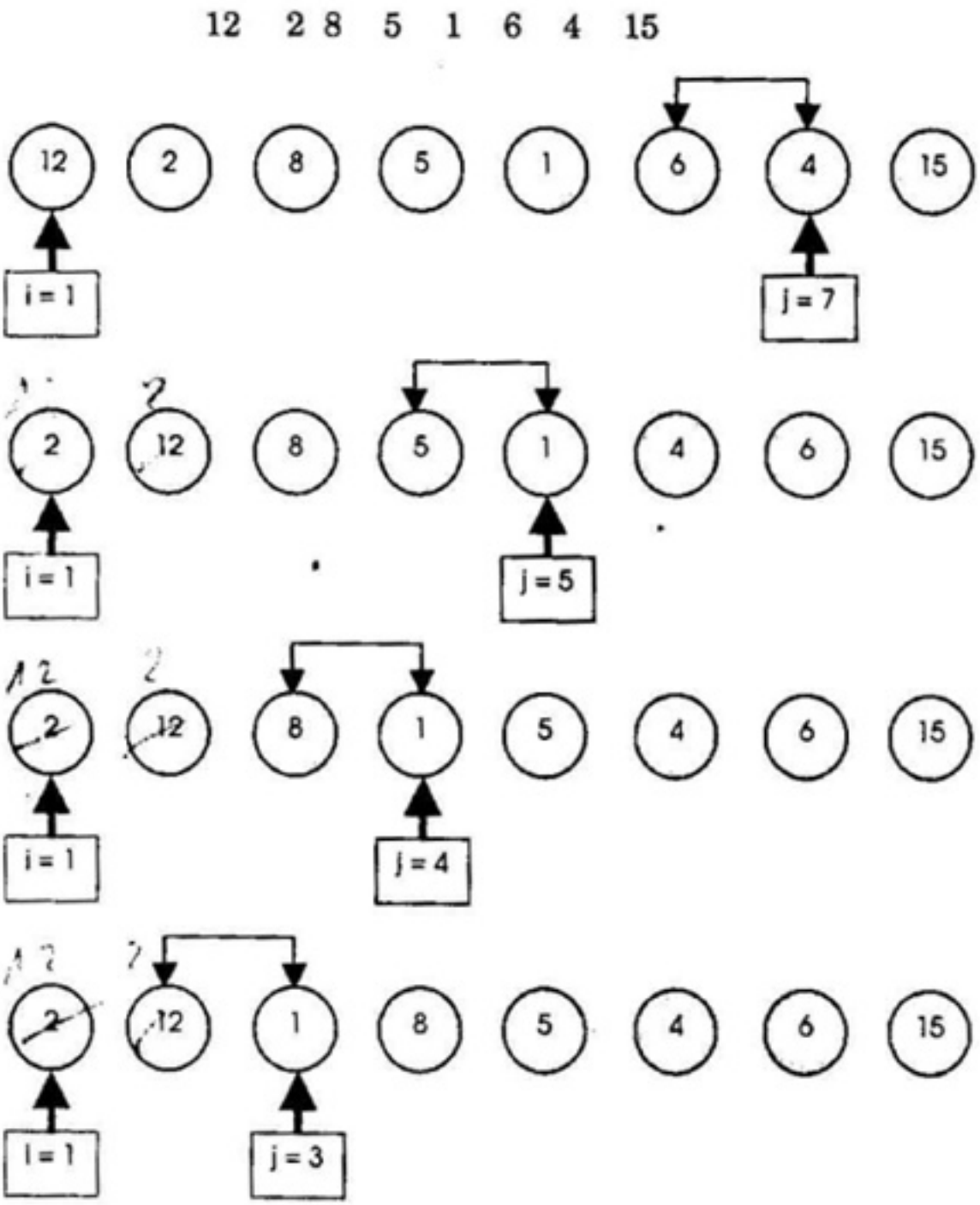
Bước 3 : i = i+1; // lần xử lý kế tiếp

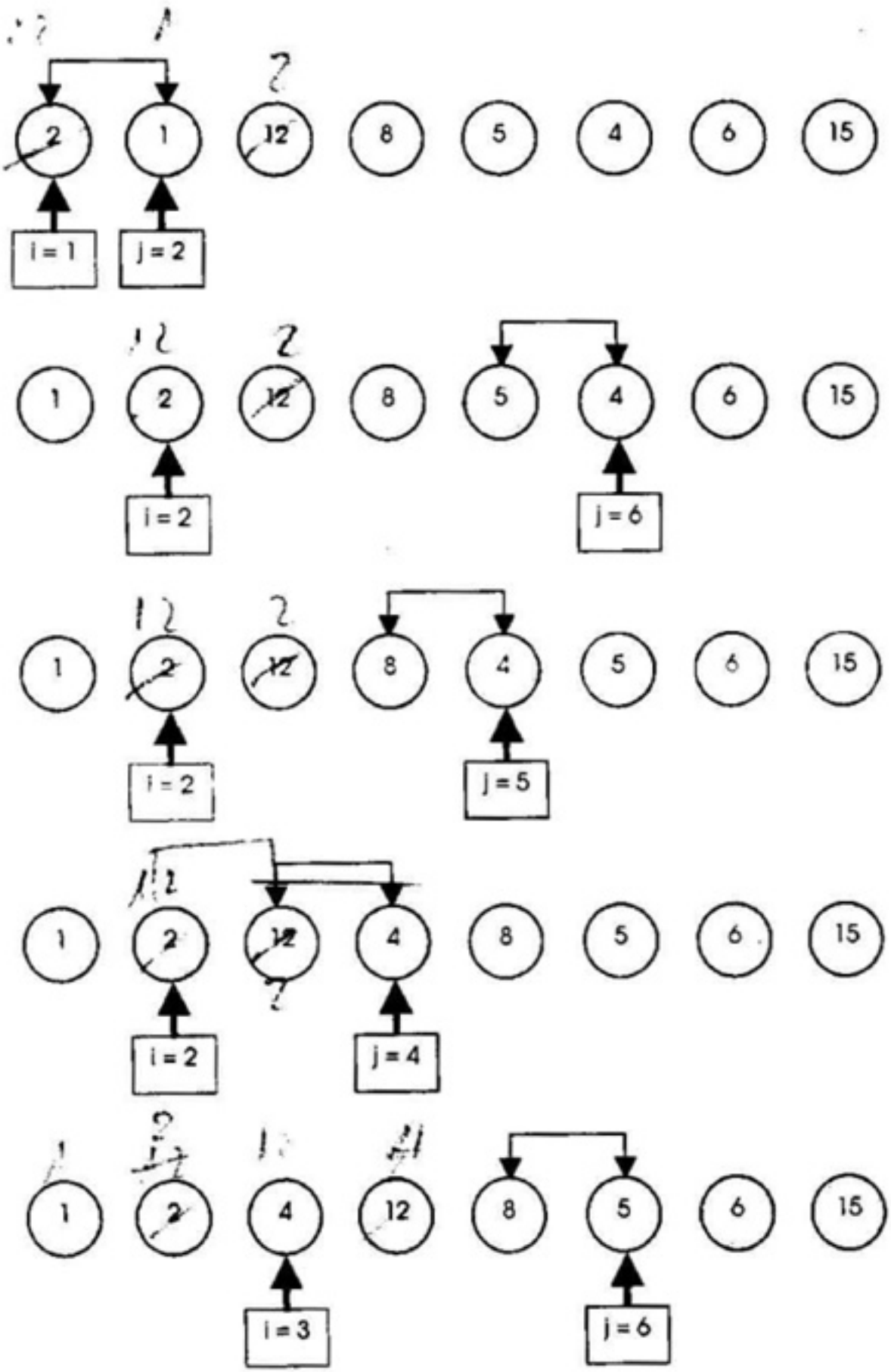
Nếu i >N-1: Hết dãy. Dừng

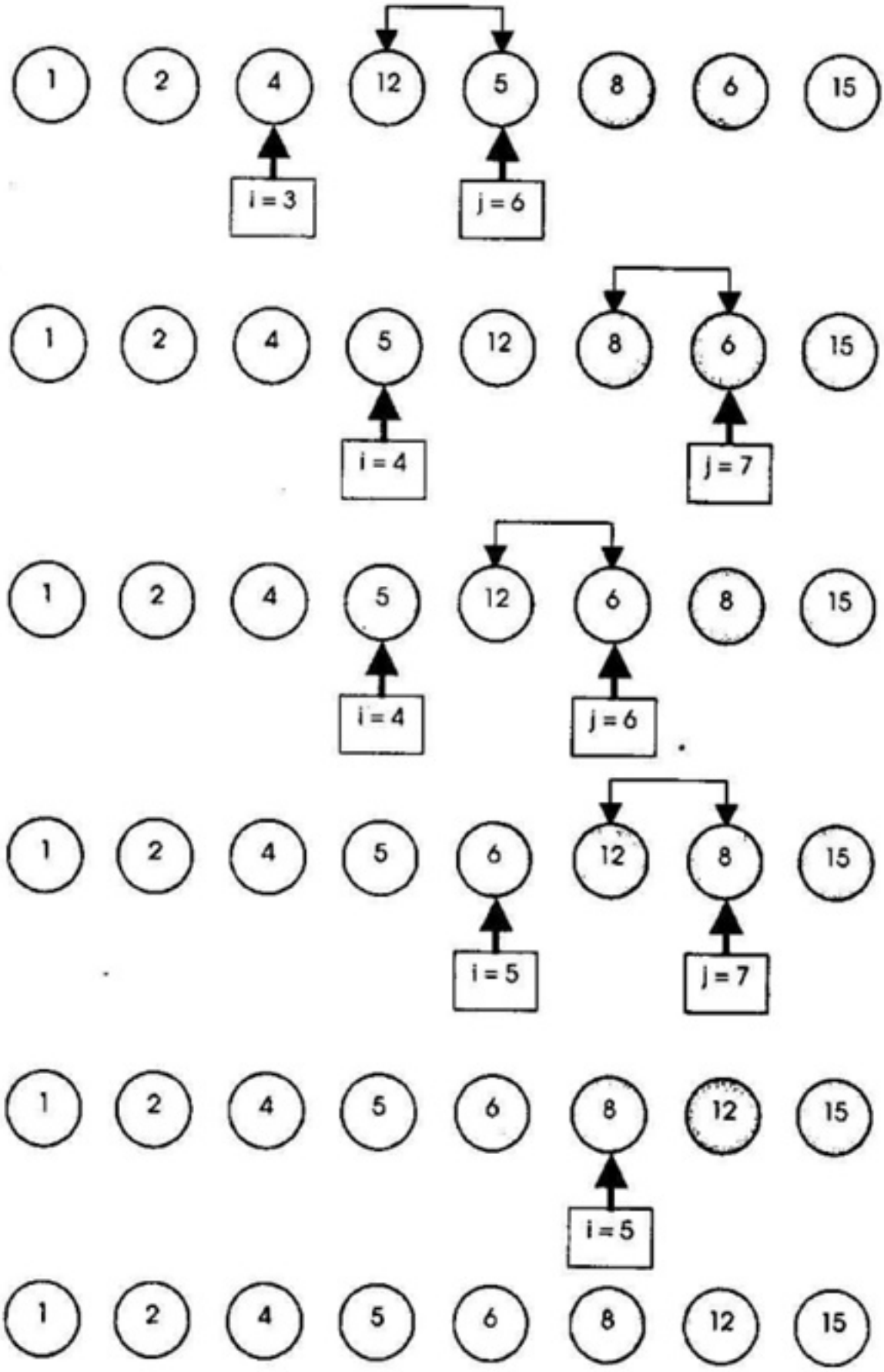
Ngược lại : Lặp lại Bước 2.

- **Ví dụ**

Cho dãy số a:




2
1
12
8
5
4
6
15
i = 1
j = 2
1
2
12
8
5
4
6
15
i = 2
j = 6
1
2
12
8
4
5
6
15
i = 2
j = 5
1
2
12
4
8
5
6
15
i = 2
j = 4
1
2
4
12
8
5
6
15
i = 3
j = 6


1 2 4 12 5 8 6 15
i = 3
j = 6
1 2 4 5 12 8 6 15
i = 4
j = 7
1 2 4 5 12 6 8 15
i = 4
j = 6
1 2 4 5 6 12 8 15
i = 5
j = 7
1 2 4 5 6 8 12 15
i = 5
1 2 4 5 6 8 12 15

- **Cài đặt**

Cài đặt thuật toán sắp xếp theo kiểu nổi bọt thành hàm BubbleSort

```
void BubleSort(int a[], int N )
{  int   i, j;
   for (i = 0 ; i<N-1 ; i++)
      for (j =N-1; j >i ; j --)
         if(a[j]< a[j-1])   // nếu sai vị trí thì đổi chỗ
            Hoanvi(a[j],a[j-1]);
}
```

- **Đánh giá giải thuật**

Đối với giải thuật nổi bọt, số lượng các phép so sánh xảy ra không phụ thuộc vào tình trạng của dãy số ban đầu, nhưng số lượng phép hoán vị thực hiện tùy thuộc vào kết quả so sánh, có thể ước lượng trong từng trường hợp như sau :

Trường hợp	Số lần so sánh	Số lần hoán vị
Tốt nhất	$\sum_{i=1}^{n-1}(n-i+1)=\frac{n(n-1)}{2}$	0
Xấu nhất	$\frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1}(n-i+1)=\frac{n(n-1)}{2}$

NHẬN XÉT

- Bubble sort có các khuyết điểm sau: không nhận diện được tình trạng dãy đã có thứ tự hay có thứ tự từng phần. Các phần tử nhỏ được đưa về vị trí đúng rất nhanh, trong khi các phần tử lớn lại được đưa về vị trí đúng rất chậm.

- Giải thuật sắp xếp **Shaker sort** cũng dựa trên nguyên tắc đổi chỗ trực tiếp, nhưng tìm cách khắc phục các nhược điểm của Buble sort với những ý tưởng cải tiến chính như sau :

Trong mỗi lần sắp xếp, duyệt mảng theo hai lượt từ hai phía khác nhau :

+ Lượt đi: đẩy phần tử nhỏ về đầu mảng

+ Lượt về: đẩy phần tử lớn về cuối mảng

Ghi nhận lại những đoạn đã sắp xếp nhằm tiết kiệm các phép so sánh thừa.

Các bước tiến hành như sau :

Bước 1 :

```
l = 1; r = n; //từ l đến r là đoạn cần được sắp xếp
k = n;  // ghi nhận vị trí k xảy ra hoán vị sau cùng
        // để làm cơ sở thu hẹp đoạn l đến r
```

Bước 2 :

```
Bước 2a : j = r ; // đẩy phần tử nhỏ về đầu mảng
    Trong khi (j > l) :
        Nếu a[j]<a[j-1]: a[j] ↔ a[j-1];
                k = j;//lưu lại nơi xảy ra hoán vị
        j = j-1;
    l = k; //loại các phần tử đã có thứ tự ở đầu dãy
Bước 2b : j = l ; // đẩy phần tử lớn về cuối mảng
    Trong khi (j < r) :
        Nếu a[j]>a[j+1]:a[j] ↔ a[j+1];
                k = j;//lưu lại nơi xảy ra hoán vị
        j = j+1;
    r = k; //loại các phần tử đã có thứ tự ở cuối dãy
```

Bước 3 : Nếu l < r: Lặp lại Bước 2.

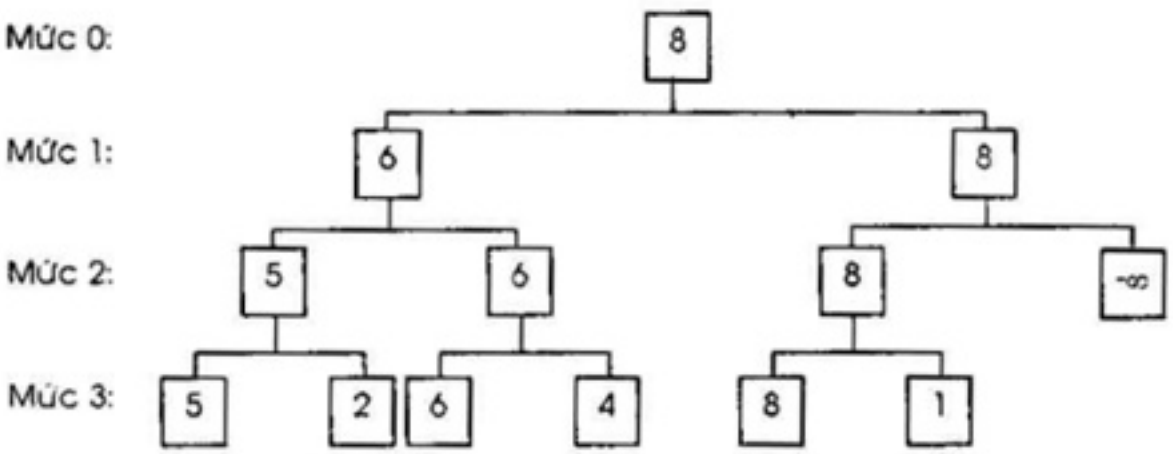
7. Sắp xếp cây - Heap sort

- **Giải thuật**

Khi tìm phần tử nhỏ nhất ở bước i, phương pháp sắp xếp chọn trực tiếp không tận dụng được các thông tin đã có được do các phép so sánh ở bước i-1. Vì lý do trên người ta tìm cách xây dựng một thuật toán sắp xếp có thể khắc phục nhược điểm này.

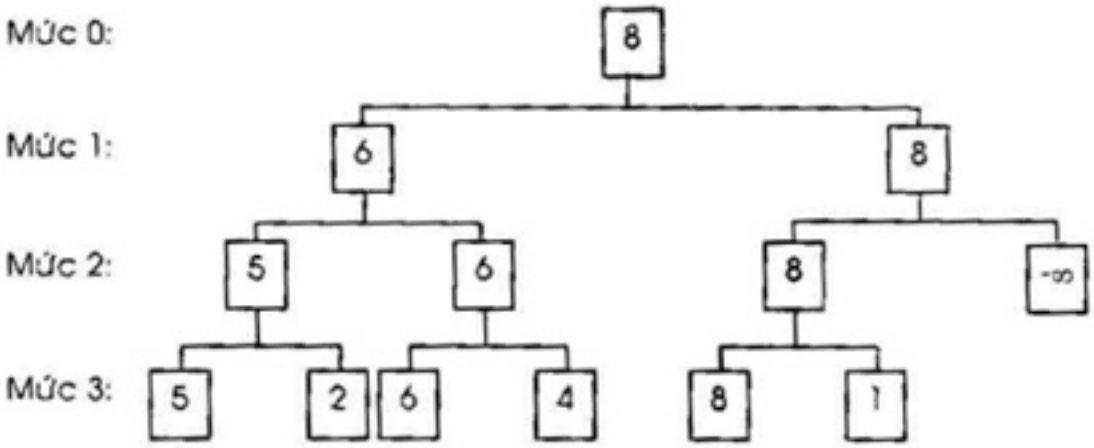
Mấu chốt để giải quyết vấn đề vừa nêu là phải tìm ra được một cấu trúc dữ liệu cho phép tích lũy các thông tin về sự so sánh giá trị các phần tử trong qua trình sắp xếp. Giả sử dữ liệu cần sắp xếp được bố trí theo quan hệ so sánh và tạo thành sơ đồ dạng cây như sau :

Dãy số : 5 2 6 4 8 1

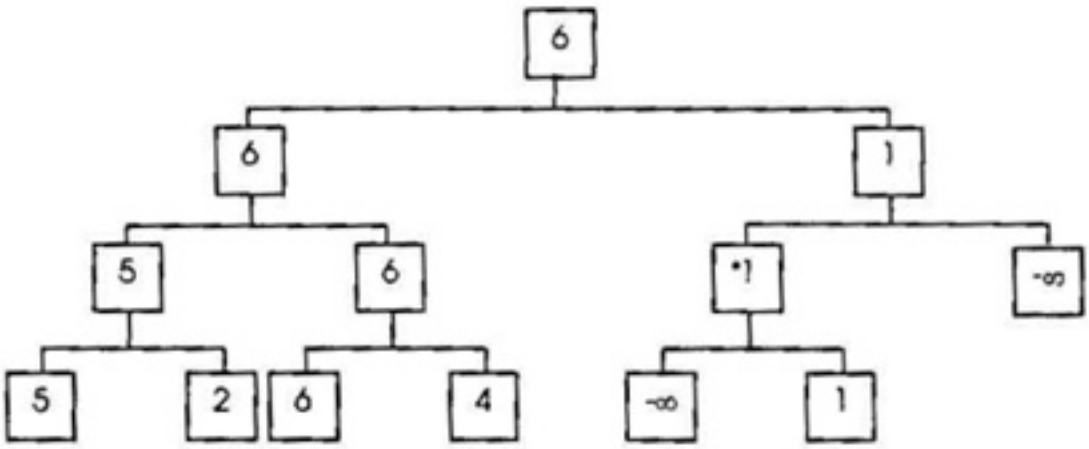


Trong đó một phần tử ở mức i chính là phần tử lớn trong cặp phần tử ở mức i+1, do đó phần tử ở mức 0 (nút gốc của cây) luôn là phần tử lớn nhất của dãy. Nếu loại bỏ phần tử gốc ra khỏi cây (nghĩa là đưa phần tử lớn nhất về đúng vị trí), thì việc cập nhật cây chỉ xảy ra trên những nhánh liên quan đến phần tử mới loại bỏ, còn các nhánh khác được bảo toàn, nghĩa là bước kế tiếp có thể sử dụng lại các kết quả so

sánh ở bước hiện tại. Trong ví dụ trên ta có :



Loại bỏ 8 ra khỏi cây và thế vào các chỗ trống giá trị -∞ để tiện việc cập nhật lại cây :



Có thể nhận thấy toàn bộ nhánh trái của gốc 1 cũ được bảo toàn, do vậy bước kế tiếp để chọn được phần tử lớn nhất hiện hành là 6, chỉ cần làm thêm một phép so sánh 1 với 6.

Tiến hành nhiều lần việc loại bỏ phần tử gốc của cây cho đến khi tất cả các phần tử của cây đều là -∞, khi đó xếp các phần tử theo thứ tự loại bỏ trên cây sẽ có dãy đã sắp xếp. Trên đây là ý tưởng của giải thuật sắp xếp cây. Tuy nhiên, để cài đặt thuật toán này một cách hiệu quả, cần phải tổ chức một cấu trúc lưu trữ dữ liệu có khả năng thể

hiện được quan hệ của các phần tử trong cây với n ô nhớ thay vì 2n-1 như trong ví dụ. Khái niệm heap và phương pháp sắp xếp Heap sort do J.Williams đề xuất đã giải quyết được các khó khăn trên.

Định nghĩa Heap

Giả sử xét trường hợp sắp xếp tăng dần, khi đó Heap được định nghĩa là một dãy các phần tử $a_l, a_2, \ldots, a_r$ thoả các quan hệ với mọi i ∈ [l, r]:

$a_i \geq a_{2i}$

$a_i \geq a_{2i+1}$ {(a_i, a_{2i}), (a_i, a_{2i+1}) là các cặp phần tử liên đới }

Heap có các tính chất sau

Tính chất 1: Nếu $a_l, a_2, \ldots, a_r$ là một heap thì khi cắt bỏ một số phần tử ở hai đầu của heap, dãy con còn lại vẫn là một heap.

Tính chất 2: Nếu $a_1, a_2, \ldots, a_n$ là một heap thì phần tử a_1 (đầu heap) luôn là phần tử lớn nhất trong heap.

Tính chất 3: Mọi dãy $a_l, a_2, \ldots, a_r$ với 2l > r là một heap.

Giải thuật Heap sort

Giải thuật Heap sort trải qua hai giai đoạn :

Giai đoạn 1: Hiệu chỉnh dãy số ban đầu thành heap;

Giai đoạn 2: Sắp xếp dãy số dựa trên heap:

Bước 1: Đưa phần tử nhỏ nhất về vị trí đúng ở cuối dãy:

r = n; Hoánvị (a_1, a_r);

Bước 2: Loại bỏ phần tử nhỏ nhất ra khỏi heap: r = r-1;

Hiệu chỉnh phần còn lại của dãy từ a_1 , a_2 ... a_r thành một heap.

Bước 3: Nếu r>1 (heap còn phần tử): Lặp lại Bước 2

Ngược lại : Dừng

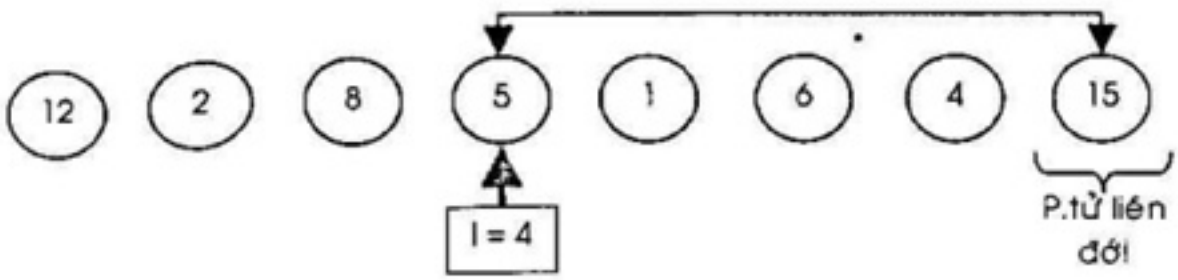
Dựa trên tính chất 3, ta có thể thực hiện giai đoạn 1 bằng cách bắt đầu từ heap mặc nhiên $a_{n/2+1}$, $a_{n/2+2}$ ... a_n, lần lượt thêm vào các phần tử $a_{n/2}$, $a_{n/2-1}$, ..., a_1 ta sẽ nhân được heap theo mong muốn. Như vậy, giai đoạn 1 tương đương với n/2 lần thực hiện bước 2 của giai đoạn 2.

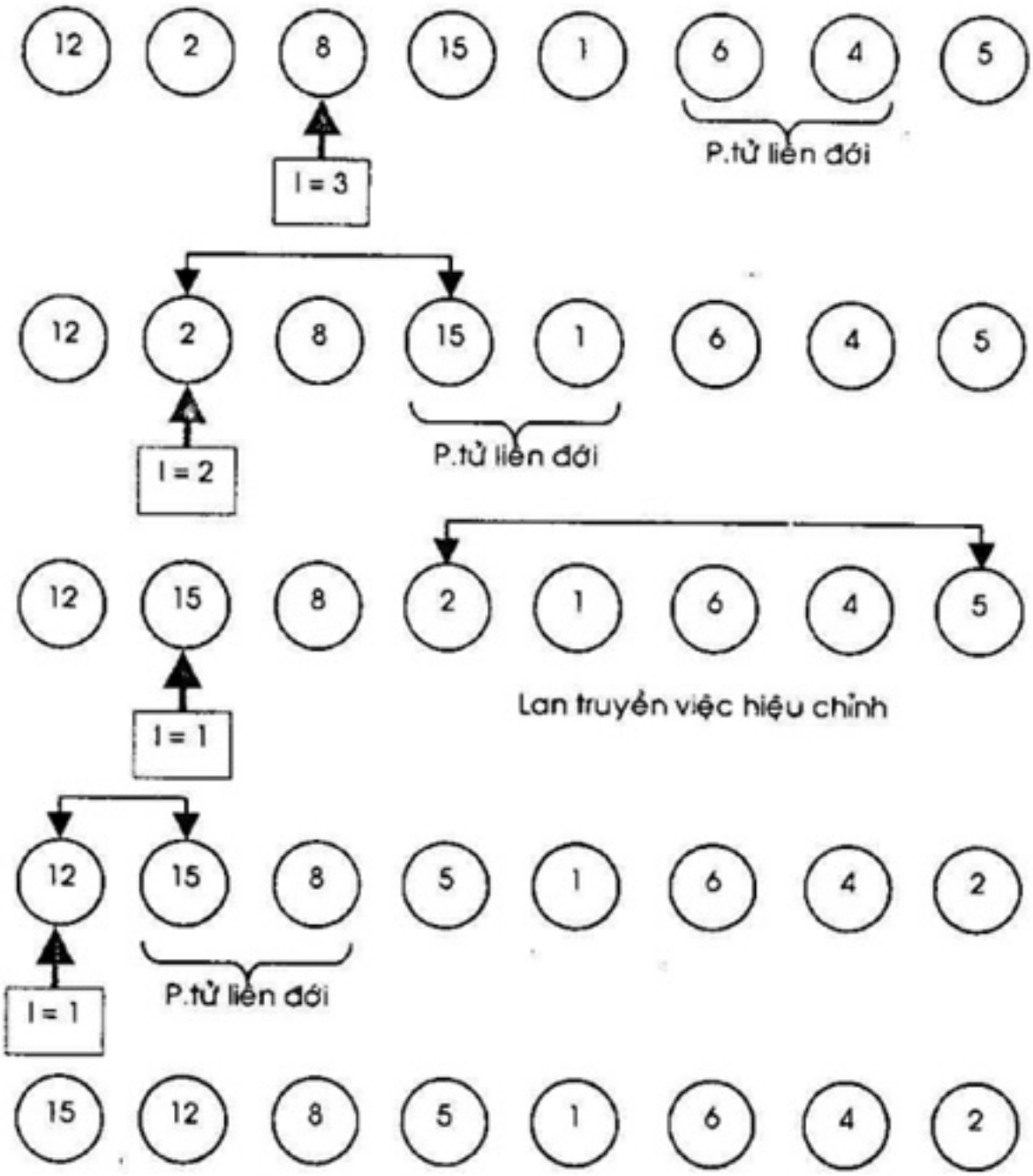
- **Ví dụ**

Cho dãy số a:

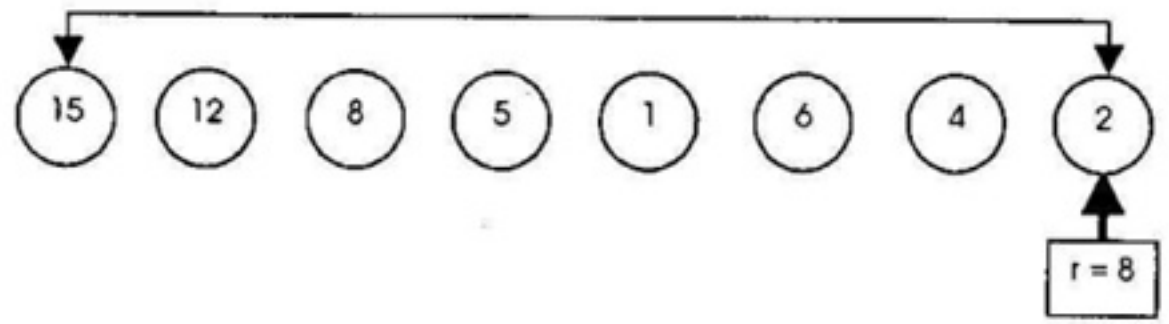
12 2 8 5 1 6 4 15

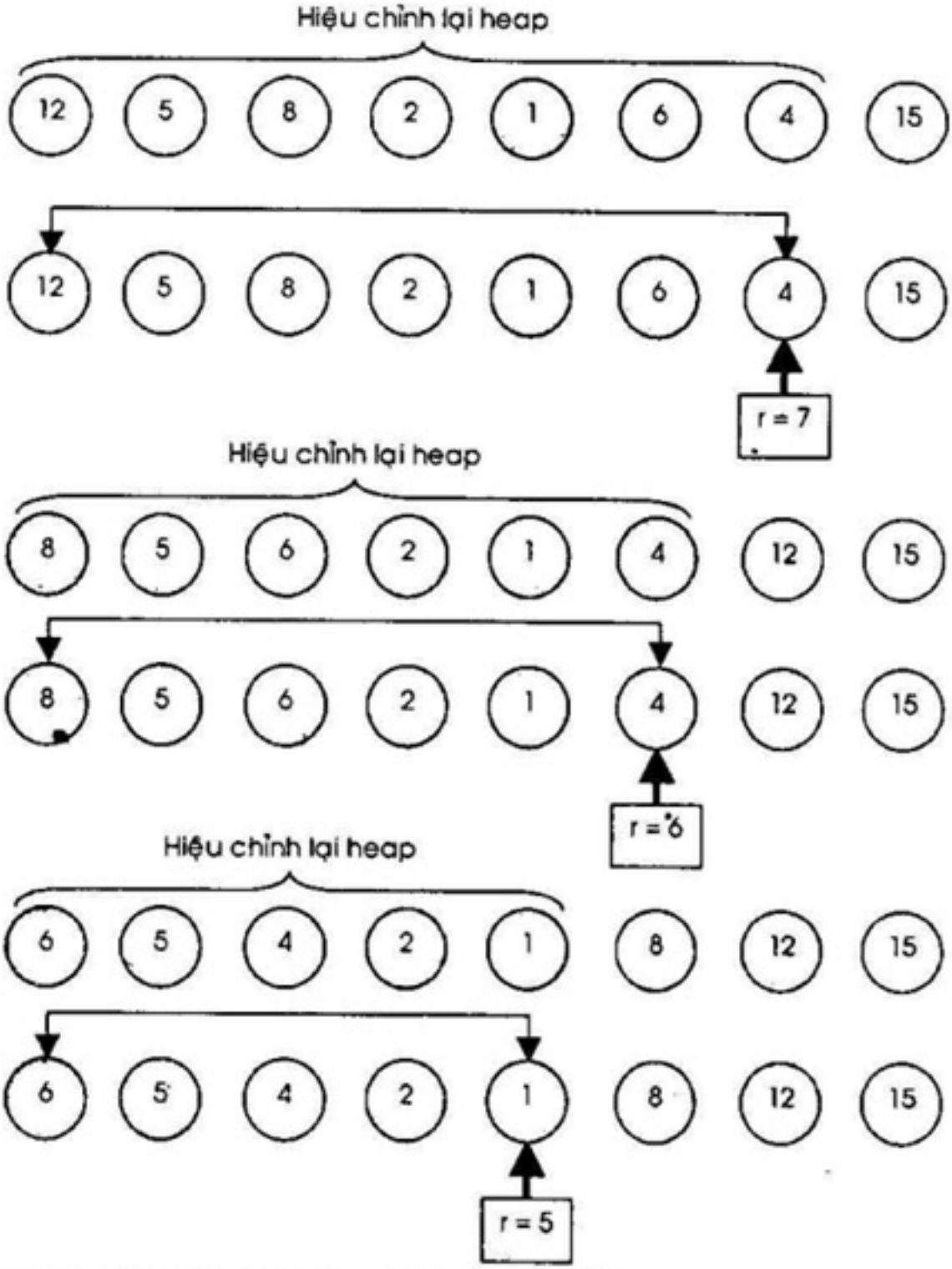
Giai đoạn 1: hiệu chỉnh dãy ban đầu thành heap





Giai đoạn 2: Sắp xếp dãy số dựa trên heap





thực hiện tương tự cho r = 5, 4, 3, 2 ta được:

- **Cài đặt**

Để cài đặt giải thuật Heap sort cần xây dựng các thủ tục phụ trợ:

1. Thủ tục hiệu chỉnh dãy a_l, a_{l+1} ...a_r thành heap

Giả sử có dãy a_l, a_{l+1} ...a_r, trong đó đoạn a_{l+1} ...a_r, đã là một heap. Ta cần xây dựng hàm hiệu chỉnh a_l, a_{l+1} ...a_r thành heap. Để làm điều này, ta lần lượt xét quan hệ của một phần tử a_i nào đó với các phần tử liên đới của nó trong dãy là a_{2i} và a_{2i+1}, nếu vi phạm điều kiện quan hệ của heap, thì đổi chỗ a_i với phần tử liên đới thích hợp của nó. Lưu ý việc đổi chỗ này có thể gây phản ứng dây chuyền:

```
void Shift (int a[ ], int l, int r )
{  int   x,i,j;
   i = l; j =2*i; // (ai, aj ). (ai, aj+1) là các phần tử liên đới
   x = a[i];
   while ((j<=r)&&(cont))
   {
   if (j<r) // nếu có đủ hai phần tử liên đới
      if (a[j]<a[j+1])// xác định phần tử liên đới lớn nhất
         j = j+1;
      if (a[j]<x)exit();// thoả quan hệ liên đới, dừng.
      else
      {  a[i] = a[j];
         i = j;    // xét tiếp khả năng hiệu chỉnh lan truyền
         j = 2*i;
         a[i] = x;
      }
   }
}
```

2. Hiệu chỉnh dãy a_1, a_2 ...a_N thành heap

Cho một dãy bất kỳ a_1, a_2, ..., a_r, theo tính chất 3, ta có dãy $a_{n/2+1}$, $a_{n/2+2}$ ... a_n đã là một heap. Ghép thêm phần tử $a_{n/2}$

vào bên trái heap hiện hành và hiệu chỉnh lại dãy $a_{n/2}$, $a_{n/2+1}$, ..., a_r thành heap, ...:

```
void CreateHeap(int a[], int N )
{  int   l;
   l = N/2; // a[l] là phần tử ghép thêm
   while (l > 0) do
   {
      Shift(a,l,N);
      l = l -1;
   }
}
```

Khi đó hàm Heapsort có dạng sau :

```
void  HeapSort (int a[], int N)
{  int   r;
   CreateHeap(a,N)
   r = N-1; // r là vị trí đúng cho phần tử nhỏ nhất
   while(r > 0) do
   {
      Hoanvi(a[l],a[r]);
      r = r -1;
      Shift(a,l,r);
   }
}
```

- **Đánh giá giải thuật**

Việc đánh giá giải thuật Heap sort rất phức tạp, nhưng đã chứng minh được trong trường hợp xấu nhất độ phức tạp ≈ $O(n\log_2 n)$.

8. Sắp xếp với độ dài bước giảm dần - Shell sort

- **Giải thuật**

Giải thuật Shell sort là một phương pháp cải tiến của

phương pháp chèn trực tiếp. Ý tưởng của phương pháp sắp xếp là phân chia dãy ban đầu thành những dãy con gồm các phần tử ở cách nhau **h** vị trí:

Dãy ban đầu : $a_1, a_2, ..., a_n$ được xem như sự xen kẽ của các dãy con sau :

Dãy con thứ nhất : $a_1\ a_{h+1}\ a_{2h+1}$ ...
Dãy con thứ hai : $a_2\ a_{h+2}\ a_{2h+2}$ ...
....
Dãy con thứ h : $a_h\ a_{2h}\ a_{3h}$ ...

Tiến hành sắp xếp các phần tử trong cùng dãy con sẽ làm cho các phần tử được đưa về vị trí đúng tương đối (chỉ đúng trong dãy con, so với toàn bộ các phần tử trong dãy ban đầu có thể chưa đúng) một cách nhanh chóng, sau đó giảm khoảng cách **h** để tạo thành các dãy con mới (tạo điều kiện để so sánh một phần tử với nhiều phần tử khác trước đó không ở cùng dãy con với nó) và lại tiếp tục sắp xếp... Thuật toán dừng khi **h** = 1, lúc này bảo đảm tất cả các phần tử trong dãy ban đầu sẽ được so sánh với nhau để xác định trật tự đúng cuối cùng.

Yếu tố quyết định tính hiệu quả của thuật toán là cách chọn khoảng cách **h** trong từng bước sắp xếp và số bước sắp xếp. Giả sử quyết định sắp xếp **k** bước, các khoảng cách chọn phải thỏa điều kiện :

$$h_i > h_{i+1} \text{ và } h_k = 1$$

Tuy nhiên đến nay vẫn chưa có tiêu chuẩn rõ ràng trong việc lựa chọn dãy giá trị khoảng cách tốt nhất, một số dãy được Knuth đề nghị :

$$h_i = (h_{i-1} - 1)/3 \text{ và } h_k = 1,\ k = \log_3 n - 1$$

Ví dụ : 127, 40, 13, 4, 1

hay

$$h_i = (h_{i-1} - 1)/2 \text{ và } h_k = 1, k = \log_2 n - 1$$

Ví dụ : 15, 7, 3, 1

Các bước tiến hành như sau:

- Bước 1 : Chọn **k** khoảng cách h[1], h[2], ..., h[k]; i = 1;

  Bước 2 :Phân chia dãy ban đầu thành các dãy con cách nhau h[i] khoảng cách. Sắp xếp từng dãy con bằng phương pháp chèn trực tiếp

  Bước 3 : i = i+1;
  Nếu i > k : Dừng
  Nếu i<=k: Lặp lại Bước 2.

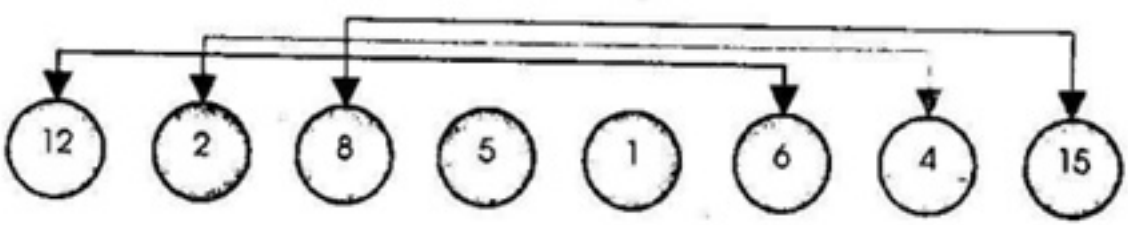
- **Ví dụ**

  Cho dãy số a:

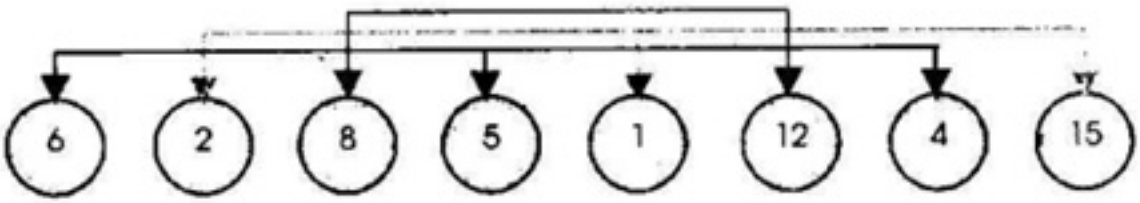
  12 2 8 5 1 6 4 15

  Giả sử chọn các khoảng cách là 5, 3, 1

  h = 5 : xem dãy ban đầu như các dãy con

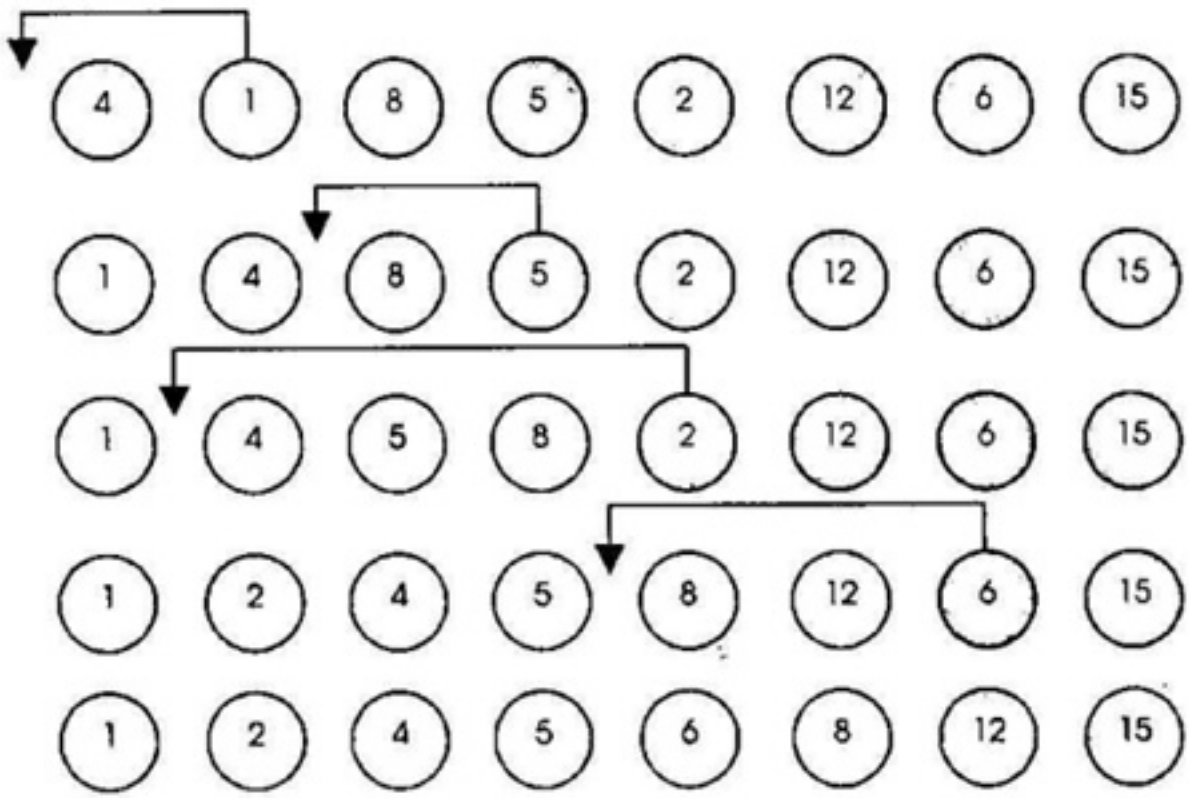


h = 3 : (sau khi đã sắp xếp các dãy con ở bước trước)



h = 1 : (sau khi đã sắp xếp các dãy con ở bước trước)

Sắp xếp dãy ta có:



Dừng .

- **Cài đặt**

Giả sử đã chọn được dãy độ dài h[1], h[2], ..., h[k], thuật toán Shell sort có thể được cài đặt như sau :

```
void ShellSort(int a[], int N, int h[], int k)
{
```

```
    intstep,i,j;
    int   x,len;

    for (step = 0 ; step <k; step ++)
    {
       len = h[step];
       for (i = len; i <N; i++)
       {  x = a[i];
          j = i-len; // a[j] đứng kề trước a[i] trong cùng dãy
          con
          while ((x<a[j])&&(j>=0)// sắp xếp dãy con
          chứa x
          {// bằng phương pháp chèn trực tiếp
             a[j+len]  = a[j];
             j = j - len;
          }
          a[j+len] = x;
       }
    }
}
```

- **Đánh giá giải thuật**

Hiện nay việc đánh giá giải thuật Shell sort dẫn đến những vấn đề toán học rất phức tạp, thậm chí một số chưa được chứng minh. Tuy nhiên hiệu quả của thuật toán còn phụ thuộc vào dãy các độ dài được chọn. Trong trường hợp chọn dãy độ dài theo công thức $h_i = (h_{i-1} - 1)/2$ và $h_k = 1$, $k = \log_2 - 1$ thì giải thuật có độ phức tạp $\approx n^{1.2} << n^2$

9. Sắp xếp dựa trên phân hoạch - Quick sort

- **Giải thuật**

Để sắp xếp dãy $a_1, a_2, ..., a_n$ giải thuật Quick sort dựa trên việc phân hoạch dãy ban đầu thành hai phần :

- *Dãy con 1*: Gồm các phần tử a_1.. a_i có giá trị không lớn hơn **x**
- *Dãy con 2*: Gồm các phần tử a_i .. a_n có giá trị không nhỏ hơn **x**

với **x** là giá trị của một phần tử tùy ý trong dãy ban đầu. Sau khi thực hiện phân hoạch, dãy ban đầu được phân thành ba phần:

1. $a_k < x$, với k = 1..i
2. $a_k = x$, với k = i..j
3. $a_k > x$, với k = j..N

$a_k < x$	$a_k = x$	$a_k > x$

trong đó dãy con thứ 2 đã có thứ tự, nếu các dãy con 1 và 3 chỉ có 1 phần tử thì chúng cũng đã có thứ tự, khi đó dãy ban đầu đã được sắp. Ngược lại, nếu các dãy con 1 và 3 có nhiều hơn 1 phần tử thì dãy ban đầu chỉ có thứ tự khi các dãy con 1, 3 được sắp. Để sắp xếp dãy con 1 và 3, ta lần lượt tiến hành việc phân hoạch từng dãy con theo cùng phương pháp phân hoạch dãy ban đầu vừa trình bày ...

Giải thuật phân hoạch dãy a_l, a_{l+1}, ..., a_r thành hai dãy con

Bước 1 : Chọn tùy ý một phần tử a[k] trong dãy là giá trị mốc, $l \le k \le r$:

```
x = a[k];  i = l;  j = r;
```

Bước 2 : Phát hiện và hiệu chỉnh cặp phần tử a[i], a[j] nằm sai chỗ :

Bước 2a: Trong khi (a[i] < x) i++;

Bước 2b: Trong khi (a[j] > x) j--;

Bước 2c: Nếu i < j// a[i] ≥ x ≥ a[j] mà a[j] đứng sau a[i]

Hoán vị (a[i],a[j]);

Bước 3 :

Nếu i < j: Lặp lại Bước 2.//chưa xét hết mảng

Nếu i ≥ j: Dừng

NHẬN XÉT

- Về nguyên tắc, có thể chọn giá trị mốc x là một phần tử tùy ý trong dãy, nhưng để đơn giản, dễ diễn đạt giải thuật, phần tử có vị trí giữa thường được chọn, khi đó k = (l +r)/ 2.
- Giá trị mốc x được chọn sẽ có tác động đến hiệu quả thực hiện thuật toán vì nó quyết định số lần phân hoạch. Số lần phân hoạch sẽ ít nhất nếu ta chon được x là phần tử median của dãy. Tuy nhiên do chi phí xác định phần tử median quá cao nên trong thực tế người ta không chọn phần tử này mà chọn phần tử nằm chính giữa dãy làm mốc với hy vọng nó có thể gần với giá trị median.

Giải thuật để sắp xếp dãy a_l, a_{l+1}, ..., a_r:

Có thể phát biểu giải thuật sắp xếp Quick sort một cách đệ qui như sau :

Bước 1

Phân hoạch dãy a_l ... a_r thành các dãy con :

- Dãy con 1 : $a_l .. a_j \leq x$
- Dãy con 2 : $a_{j+1} .. a_{i-1} = x$
- Dãy con 1 : $a_i .. a_r \geq x$

Bước 2

Nếu (l < j) // dãy con 1 có nhiều hơn 1 phần .

Phân hoạch dãy a_l.. a_j

Nếu (i < r) // dãy con 3 có nhiều hơn 1 phần tử

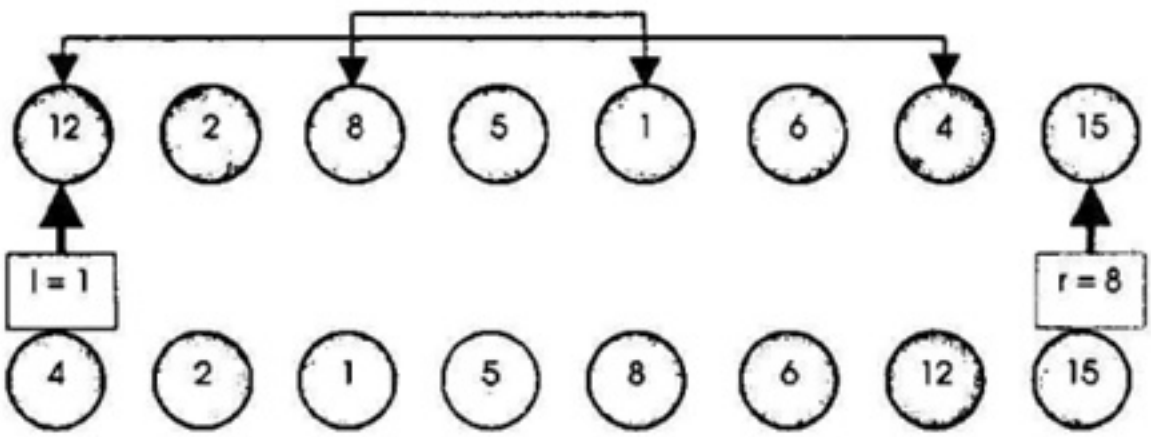
Phân hoạch dãy a_i.. a_r

- **Ví dụ**

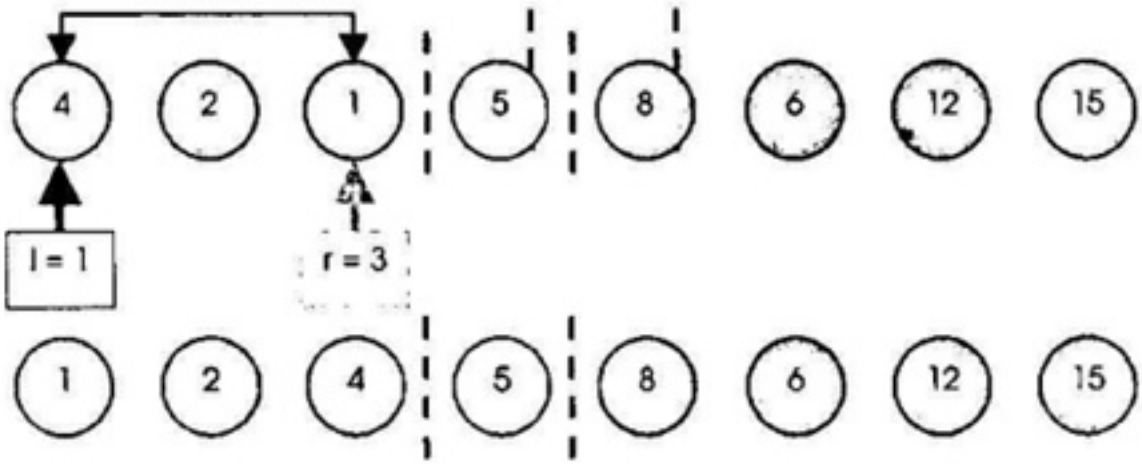
Cho dãy số a:

12 2 8 5 1 6 4 15

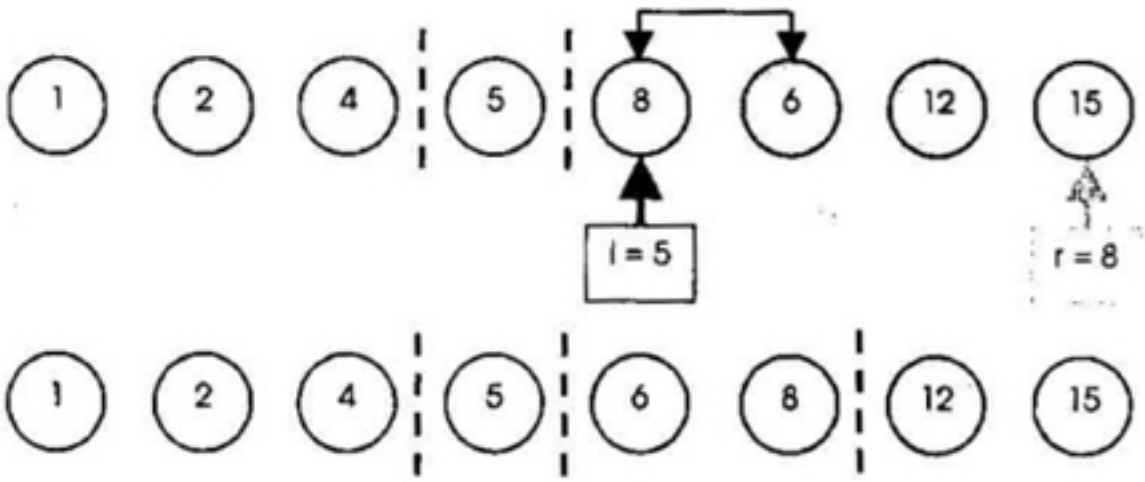
Phân hoạch đoạn l =1, r = 8: **x = A[4] = 5**



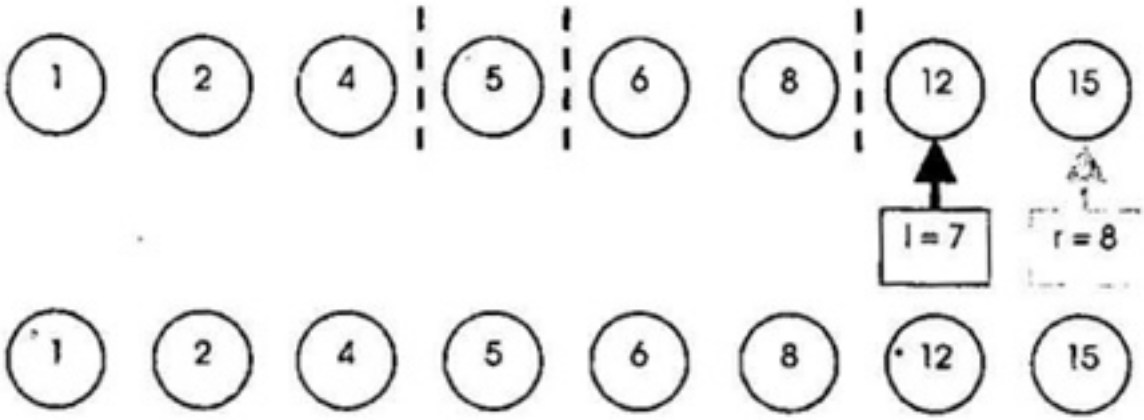
Phân hoạch đoạn l =1, r = 3: **x = A[2] = 2**



Phân hoạch đoạn l = 5, r = 8: **x = A[6] = 6**



Phân hoạch đoạn l = 7, r = 8: **x = A[7] = 6**



Dừng.

- **Cài đặt**

Thuật toán Quick sort có thể được cài đặt đệ qui như sau :

```
void QuickSort(int a[], int l, int r)
{
   int   i,j;
   int   x;
   x = a[(l+r)/2];     // chọn phần tử giữa làm giá trị mốc
   i =l; j = r;
   do {
   while(a[i] < x) i++;
   while(a[j] > x) j--;
   if(i <= j)
```

```
    {
       Hoanvi(a[i],a[j]);
       i++ ; j--;
    }
    }while(i < j);
    if(l < j)
    QuickSort(a,l,j);
    if(i < r)
    QuickSort(a,i,r);
}
```

- **Đánh giá giải thuật**

Hiệu quả thực hiện của giải thuật Quick sort phụ thuộc vào việc chọn giá trị mốc. Trường hợp tốt nhất xảy ra nếu mỗi lần phân hoạch đều chọn được phần tử median (phần tử lớn hơn (hay bằng) nửa số phần tử, và nhỏ hơn (hay bằng) nửa số phần tử còn lại) làm mốc, khi đó dãy được phân chia thành hai phần bằng nhau và cần $\log_2(n)$ lần phân hoạch thì sắp xếp xong. Nhưng nếu mỗi lần phân hoạch lại chọn nhầm phần tử có giá trị cực đại (hay cực tiểu) là mốc, dãy sẽ bị phân chia thành hai phần không đều: một phần chỉ có một phần tử, phần còn lại gồm (n-1) phần tử, do vậy cần phân hoạch n lần mới sắp xếp xong. Ta có bảng tổng kết

Trường hợp	**Độ phức tạp**
Tốt nhất	n*log(n)
Trung bình	n*log(n)
Xấu nhất	n^2

10. Sắp xếp theo phương pháp trộn trực tiếp - Merge sort

- **Giải thuật**

Để sắp xếp dãy $a_1, a_2, ..., a_n$, giải thuật Merge sort dựa trên

nhận xét sau:

- Mỗi dãy $a_1, a_2, ..., a_n$ bất kỳ đều có thể coi như là một tập hợp các dãy con liên tiếp mà mỗi dãy con đều đã có thứ tự. Ví dụ dãy 12, 2, 8, 5, 1, 6, 4, 15 có thể coi như gồm 5 dãy con không giảm (12); (2, 8); (5); (1, 6); (4, 15).
- Dãy đã có thứ tự coi như có 1 dãy con.

Như vậy, một cách tiếp cận để sắp xếp dãy là tìm cách làm giảm số dãy con không giảm của nó. Đây chính là hướng tiếp cận của thuật toán sắp xếp theo phương pháp trộn.

Trong phương pháp Merge sort, mấu chốt của vấn đề là cách phân hoạch dãy ban đầu thành các dãy con. Sau khi phân hoạch xong, dãy ban đầu sẽ được tách ra thành hai dãy phụ theo nguyên tắc phân phối đều luân phiên. Trộn từng cặp dãy con của hai dãy phụ thành một dãy con của dãy ban đầu, ta sẽ nhân lại dãy ban đầu nhưng với số lượng dãy con ít nhất giảm đi một nửa. Lặp lại qui trình trên sau một số bước, ta sẽ nhận được một dãy chỉ gồm một dãy con không giảm; nghĩa là dãy ban đầu đã được sắp xếp.

Giải thuật trộn tự nhiên là phương pháp trộn đơn giản nhất. Việc phân hoạch thành các dãy con đơn giản chỉ là tách dãy gồm n phần tử thành n dãy con. Đòi hỏi của thuật toán về tính có thứ tự của các dãy con luôn được thỏa trong cách phân hoạch này vì dãy gồm một phần tử luôn có thứ tự. Cứ mỗi lần tách rồi trộn, chiều dài của các dãy con sẽ được nhân đôi.

Các bước thực hiện thuật toán như sau:

Bước 1 : // Chuẩn bị
k = 1; // k là chiều dài của dãy con trong bước hiện hành

Bước 2 :

Tách dãy $a_1, a_2, ..., a_n$ thành hai dãy b, c theo nguyên tắc luân phiên từng nhóm k phần tử:

$b = a_1, ..., a_k, a_{2k+1}, ..., a_{3k}, ...$

$c = a_{k+1}, ..., a_{2k}, a_{3k+1}, ..., a_{4k}, ...$

Bước 3 :

Trộn từng cặp dãy con gồm k phần tử của hai dãy b, c vào a.

Bước 4 :

k = k*2;

Nếu k < n thì trở lại bước 2.

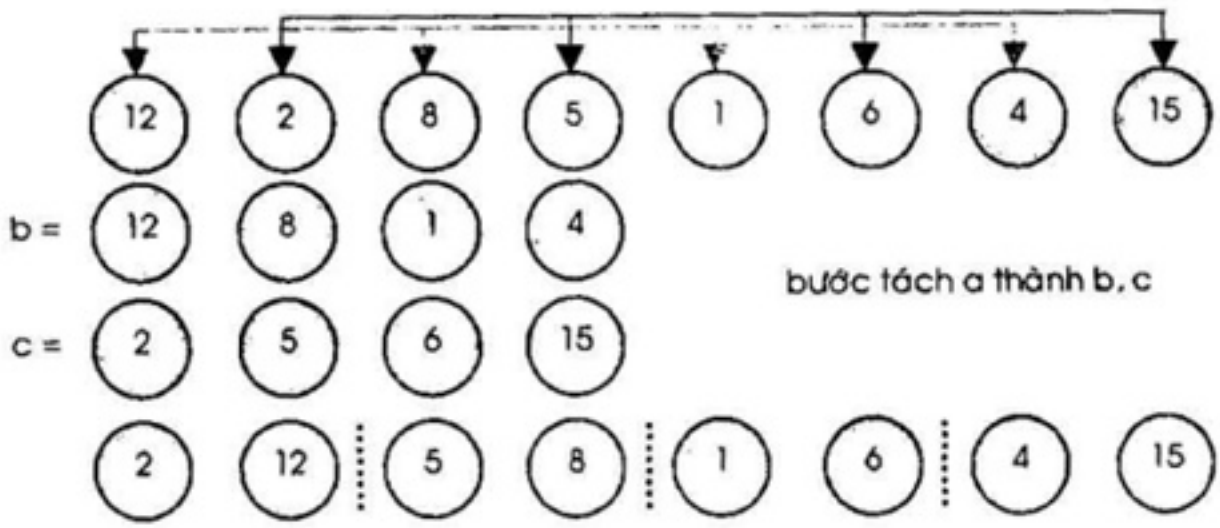
Ngược lại: Dừng

- **Ví dụ**

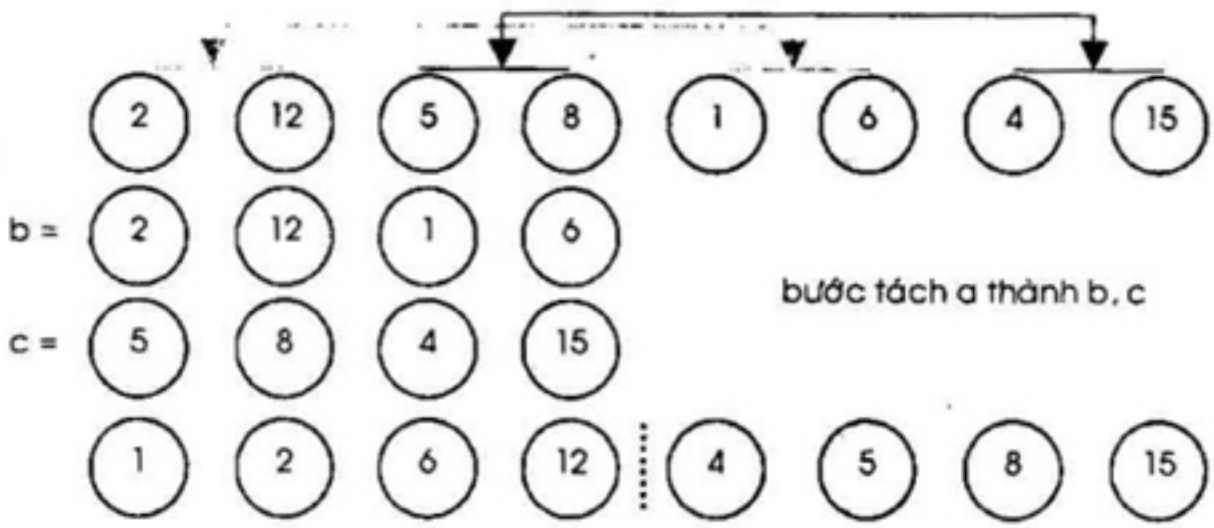
Cho dãy số a:

12 2 8 5 1 6 4 15

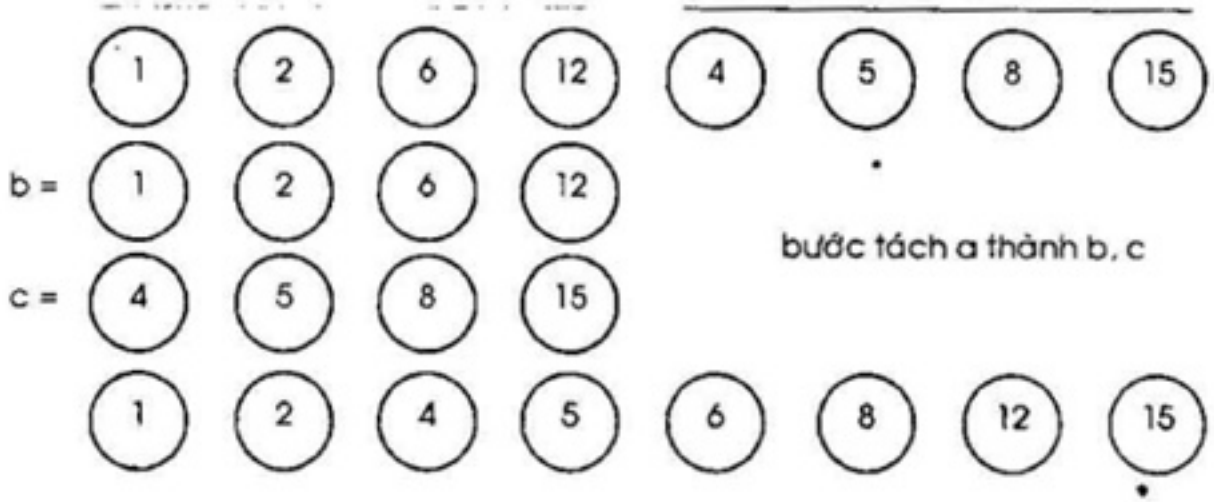
k = 1



k = 2



k = 4



- **Cài đặt**

Thuật toán Merge sort có thể được cài đặt như sau :

```
int   b[MAX], c[MAX]; // hai mảng phụ
void MergeSort(int a[], int n)
{
    int   p, pb, pc;    // các chỉ số trên các mảng a, b, c
    int   i, k = 1;     // độ dài của dãy con khi phân hoạch
    do {
        // tách a thành b và c;
```

```
        p = pb = pc = 0;
        while(p < n)   {
           for(i = 0; (p < n)&&(i < k); i++)
              b[pb++] = a[p++];
           for(i = 0; (p < n)&&(i < k); i++)
              c[pc++] = a[p++];
        }
        Merge(a, pb, pc, k); //trộn b, c lại thành a
        k *= 2;
     }while(k < n);
}
```

trong đó hàm Merge có thể được cài đặt như sau :

```
void Merge(int a[], int nb, int nc, int k)
{  int p, pb, pc, ib, ic, kb, kc;
   p = pb = pc = 0; ib = ic = 0;
   while((0 < nb)&&(0 < nc))  {
      kb = min(k, nb); kc = min(k, nc);
      if(b[pb+ib] <= c[pc+ic]){
         a[p++] = b[pb+ib]; ib++;
         if(ib == kb) {
            for(; ic<kc; ic++) a[p++] =
            c[pc+ic];
            pb += kb; pc += kc; ib = ic = 0;
            nb -= kb; nc -= kc;
         }
      }
      else  {
         a[p++] = c[pc+ic]; ic++;
         if(ic == kc) {
            for(; ib<kb; ib++) a[p++] =
            b[pb+ib];
            pb += kb; pc += kc; ib = ic = 0;
            nb -= kb; nc -= kc;
         }
      }
   }
}
```

- **Đánh giá giải thuật**

Ta thấy rằng số lần lặp của bước 2 và bước 3 trong thuật toán Merge sort bằng $\log_2 n$ do sau mỗi lần lặp giá trị của k

tăng lên gấp đôi. Dễ thấy, chi phí thực hiện bước 2 và bước 3 tỉ lệ thuận với n. Như vậy, chi phí thực hiện của giải thuật Merge sort sẽ là $O(n\log_2 n)$. Do không sử dụng thông tin nào về đặc tính của dãy cần sắp xếp, nên trong mọi trường hợp của thuật toán chi phí là không đổi. Đây cũng chính là một trong những nhược điểm lớn của thuật toán.

NHẬN XÉT

- Như trong phần đánh giá giải thuật, một trong những nhược điểm lớn của thuật toán là không tận dụng những thông tin về đặc tính của dãy cần sắp xếp. Ví dụ trường hợp dãy đã có thứ tự sẵn. Chính vì vậy, trong thực tế người ta ít dùng thuật toán trộn trực tiếp mà người ta dùng phiên bản cải tiến của nó. Phiên bản này thường được biết với tên gọi thuật toán trộn tự nhiên (Natural Merge sort).

- Để khảo sát thuật toán trộn tự nhiên, trước tiên ta cần định nghĩa khái niệm đường chạy (run):

  Một đường chạy của dãy số a là một dãy con không giảm của cực đại của a; nghĩa là, đường chạy $r = (a_i, a_{i+1}, ..., a_j)$ *phải thỏa điều kiện:*

$$\begin{cases} a_k \le a_{k+1} & \forall k \in [i, j) \\ a_i < a_{i-1} \\ a_j > a_{j+1} \end{cases}$$

  Ví du dãy 12, 2, 8, 5, 1, 6, 4, 15 có thể coi như gồm 5 đường chạy (12); (2, 8); (5); (1, 6); (4, 15).

  Thuật toán trộn tự nhiên khác thuật toán trộn trực tiếp ở chỗ thay vì luôn cứng nhắc phân hoạch theo dãy con có chiều dài k, việc phân hoạch sẽ theo đơn vị là đường

chạy. ta chỉ cần biết số đường chạy của a sau lần phân hoạch cuối cùng là có thể biết thời điểm dừng của thuật toán vì dãy đã có thứ tự là dãy chỉ có một đường chạy.

Các bước thực hiện thuật toán trộn tự nhiên như sau

Bước 1 : // Chuẩn bị

r = 0; // r dùng để đếm số đường chạy

Bước 2 :

Tách dãy $a_1, a_2, ..., a_n$ thành hai dãy b, c theo nguyên tắc luân phiên từng đường chạy:

Bước 2.1 :

Phân phối cho b một đường chạy; r = r+1;

Nếu a còn phần tử chưa phân phối

Phân phối cho c một đường chạy; r = r+1;

Bước 2.2 :

Nếu a còn phần tử: quay lại bước 21;

Bước 3 :

Trộn từng cặp đường chạy của hai dãy b, c vào a.

Bước 4 :

Nếu r <= 2 thì trở lại bước 2;

Ngược lại: Dừng;

- Một nhược điểm lớn nữa của thuật toán trộn là khi cài đặt thuật toán đòi hỏi thêm không gian bộ nhớ để lưu các dãy phụ b, c. Hạn chế này khó chấp nhận trong thực tế vì các dãy cần sắp xếp thường có kích thước lớn. Vì vậy thuật toán trộn thường được dùng để sắp xếp các cấu trúc dữ liệu khác phù hợp hơn như danh sách liên kết hoặc file. Chương sau ta sẽ gặp lại thuật toán này.

11. Sắp xếp theo phương pháp cơ số - Radix sort

- **Giải thuật**

Khác với các thuật toán trước, Radix sort là một thuật toán tiếp cận theo một hướng hoàn toàn khác. Nếu như trong các thuật toán khác, cơ sở để sắp xếp luôn là việc so sánh giá trị của hai phần tử thì Radix sort lại dựa trên nguyên tắc phân loại thư của bưu điện. Vì lý do đó nó còn có tên là Postman's sort. Nó không hề quan tâm đến việc so sánh giá trị của phần tử và bản thân việc phân loại và trình tự phân loại sẽ tạo ra thứ tự cho các phần tử.

Ta biết rằng, để chuyển một khối lượng thư lớn đến tay người nhận ở nhiều địa phương khác nhau, bưu điện thường tổ chức một hệ thống phân loại thư phân cấp. Trước tiên, các thư đến cùng một tỉnh, thành phố sẽ được sắp chung vào một lô để gửi đến tỉnh thành tương ứng. Bưu điện các tỉnh thành này lại thực hiện công việc tương tự. Các thư đến cùng một quận, huyện sẽ được xếp vào chung một lô và gửi đến quận, huyện tương ứng. Cứ như vậy, các bức thư sẽ được trao đến tay người nhận một cách có hệ thông mà công việc sắp xếp thư không quá nặng nhọc.

Mô phỏng lại qui trình trên, để sắp xếp dãy $a_1, a_2, ..., a_n$, giải thuật Radix sort thực hiện như sau:

- Trước tiên, ta có thể giả sử mỗi phần tử a_i trong dãy $a_1, a_2, ..., a_n$ là một số nguyên có tối đa m chữ số.
- Ta phân loại các phần tử lần lượt theo các chữ số hàng đơn vị, hàng chục, hàng trăm, ... tương tự việc phân loại thư theo tỉnh thành, quận huyện, phường xã, ...

Các bước thực hiện thuật toán như sau

Bước 1 : // k cho biết chữ số dùng để phân loại hiện hành
k = 0; // k = 0: hàng đơn vị; k = 1: hàng chục; ...

Bước 2 : //Tạo các lô chứa các loại phần tử khác nhau
Khởi tạo 10 lô B_0, B_1, ..., B_9 rỗng;

Bước 3 : For i = 1 .. n do
Đặt a_i vào lô B_t với t = chữ số thứ k của a_i;

Bước 4 : Nối B_0, B_1, ..., B_9 lại (theo đúng trình tự) thành a.

Bước 5 : k = k+1;
Nếu k < m thì trở lại bước 2.
Ngược lại: Dừng

- **Ví dụ**

Cho dãy số a:

701 1725 999 9170 3252 4518 7009 1424 428 1239 8425 7013

Phân lô theo hàng đơn vị:

12	0701										
11	1725										
10	0999										
9	9170										
8	3252										
7	4518										
6	7009										
5	1424										
4	0428										
3	1239										0999
2	8425						1725			4518	7009
1	7013	9170	0701	3252	7013	1424	8425			0428	1239
CS	A	0	1	2	3	4	5	6	7	8	9

Các lô B dùng để phân loại

Phân lô theo hàng chục:

12	0999										
11	7009										
10	1239										
9	4518										
8	0428										
7	1725										
6	8425										
5	1424										
4	7013			0428							
3	3252			1725							
2	0701	7009	4518	8425							
1	9170	0701	7013	1424	1239		3252		9170		0999
CS	A	0	1	2	3	4	5	6	7	8	9

Phân lô theo hàng trăm:

12	0999										
11	9170										
10	3252										
9	1239										
8	0428										
7	1725										
6	8425										
5	1424										
4	4518										
3	7013					0428					
2	7009	7013		3252		8425			1725		
1	0701	7009	9170	1239		1424	4518		0701		0999
CS	A	0	1	2	3	4	5	6	7	8	9

Phân lô theo hàng ngàn:

12	0999										
11	1725										
10	0701										
9	4518										
8	0428										
7	8425										
6	1424										
5	3252										
4	1239										
3	9170	0999	1725								
2	7013	0701	1424						7013		
1	7009	0428	1239		3252	4518			7009	8425	9170
CS	A	0	1	2	3	4	5	6	7	8	9

Lấy các phần tử từ các lô B_0, B_1, ..., B_9 nối lại thành a:

12	9170										
11	8425										
10	7013										
9	7009										
8	4518										
7	3252										
6	1725										
5	1424										
4	1239										
3	0999										
2	0701										
1	0428										
CS	A	0	1	2	3	4	5	6	7	8	9

- **Đánh giá giải thuật**

Với một dãy n số, mỗi số có tối đa m chữ số, thuật toán thực hiện m lần các thao tác phân lô và ghép lô. Trong thao tác phân lô, mỗi phần tử chỉ được xét đúng một lần, khi ghép cũng vậy. Như vậy, chi phí cho việc thực hiện thuật toán hiển nhiên là O(2mn) = O(n).

NHẬN XÉT

- Sau lần phân phối thứ k các phần tử của A vào các lô B_0, B_1, ..., B_9, và lấy ngược trở ra, nếu chỉ xét đến k+1 chữ số của các phần tử trong A, ta sẽ có một mảng tăng dần nhờ trình tự lấy ra từ 0 → 9. Nhận xét này bảo đảm tính đúng đắn của thuật toán.
- Thuật toán có độ phức tạp tuyến tính nên hiệu quả khi sắp dãy có rất nhiều phần tử, nhất là khi khóa sắp xếp không quá dài so voiứ số lượng phần tử (điều này thường gặp trong thực tế).
- Thuật toán không có trường hợp xấu nhất và tốt nhất. Mọi dãy số đều được sắp với chi phí như nhau nếu chúng có cùng số phần tử và các khóa có cùng chiều dài.
- Thuật toán cài đặt thuận tiện với các mảng với khóa sắp xếp là chuỗi (ký tự hay số) hơn là khóa số như trong ví dụ do tránh được chi phí lấy các chữ số của từng số.
- Tuy nhiên, số lượng lô lớn (10 khi dùng số thập phân, 26 khi dùng chuỗi ký tự tiếng Anh, ...) nhưng tổng kích thước của tất cả các lô chỉ bằng dãy ban đầu nên ta không thể dùng mảng để biểu diễn B. Như vậy, phải dùng cấu trúc dữ liệu động để biểu diễn B ⇒ Radix sort rất thích hợp cho sắp xếp trên danh sách liên kết.

- Người ta cũng dùng phương pháp phân lô theo biểu diễn nhị phân của khóa sắp xếp. Khi đó ta có thể dùng hoàn toàn cấu trúc dữ liệu mảng để biểu diễn B vì chỉ cần dùng hai lô B_0 và B_1. Tuy nhiên, khi đó chiều dài khóa sẽ lớn. Khi sắp các dãy không nhiều phần tử, thuật toán Radix sort sẽ mất ưu thế so với các thuật toán khác.

TÓM TẮT

Trong chương này, chúng ta đã xem xét các thuật toán tìm kiếm và sắp xếp thông dụng. Cấu trúc dữ liệu chính để minh họa các thao tác này chủ yếu là mảng một chiều. Đây cũng là một trong những cấu trúc dữ liệu thông dụng nhất.

Khi khảo sát các thuật toán tìm kiếm, chúng ta đã làm quen với hai thuật toán. Thuật toán thứ nhất là thuật toán tìm kiếm tuần tự. Thuật toán này có độ phức tạp tuyến tính (O(n)). Ưu điểm của nó là tổng quát và có thể mở rộng để thực hiện các bài toán tìm kiếm đa dạng. Tuy nhiên, chi phí thuật toán khá cao nên ít khi được sử dụng. Thuật toán thứ hai là thuật toán nhị phân tìm kiếm. Thuật toán này có ưu điểm là tìm kiếm rất nhanh (độ phức tạp là $\log_2 n$), nhưng chỉ có thể áp dụng đối với dữ liệu đã có thứ tự theo khóa tìm kiếm. Do đòi hỏi của thực tế, thao tác tìm kiếm phải nhanh vì đây là thao tác có tần suất sử dụng rất cao nên thuật toán nhị phân tìm kiếm thường được dùng hơn thuật toán tìm tuần tự. Chính vì vậy xuất hiện nhu cầu phát triển các thuật toán sắp xếp hiệu quả.

Phần tiếp theo của chương trình bày các thuật toán sắp xếp thông dụng theo thứ tự từ đơn giản đến phức tạp (từ chi phí cao đến chi phí thấp).

Phần lớn các thuật toán sắp xếp cơ bản dựa trên sự so sánh giá trị giữa các phần tử. Bắt đầu từ nhóm các thuật toán cơ bản, đơn giản nhất. Đó là các thuật toán chọn trực tiếp, chèn trực tiếp, nổi bọt, đổi chỗ trực tiếp. Các thuật toán này đều có một điểm chung là chi phí thực hiện chúng tỉ lệ với n^2.

Tiếp theo, chúng ta khảo sát một số cải tiến của các thuật toán trên.

Nếu như các thuật toán chèn nhị phân (cải tiến của chèn trực tiếp), Shaker sort (cải tiến của nổi bọt), ... tuy chi phí có ít hơn các thuật toán gốc nhưng chúng vẫn chỉ là các thuật toán thuộc nhóm có độ phức tạp $O(n^2)$, thì các thuật toán Shell sort (cải tiến của chèn trực tiếp), Heap sort (cải tiến của chọn trực tiếp) lại có độ phức tạp nhỏ hơn hẳn các thuật toán gốc. Thuật toán Shell sort có độ phức tạp $O(n^x)$ với $1 < x < 2$ và thuật toán Heap sort có độ phức tạp $O(n\log_2 n)$.

Các thuật toán Merge sort và Quick sort là những thuật toán thực hiện theo chiến lược chia để trị. Cài đặt chúng tuy phức tạp hơn các thuật toán khác nhưng chi phí thực hiện lại thấp. Cả hai thuật toán đều có độ phức tạp $O(n\log_2 n)$. Merge sort có nhược điểm là cần dùng thêm bộ nhớ đệm. Thuật toán này sẽ phát huy tốt ưu điểm của mình hơn khi cài đặt trên các cấu trúc dữ liệu khác phù hợp hơn như danh sách liên kết hay file.

Thuật toán Quick sort, như tên gọi của mình được đánh giá là thuật toán sắp xếp nhanh nhất trong số các thuật toán sắp xếp dựa trên nền tảng so sánh giá trị của các phần tử. Tuy có chi phí trong trường hợp xấu nhất là $O(n^2)$ nhưng trong kiểm nghiệm thực tế, thuật toán Quick sort chạy nhanh hơn hai thuật toán cùng nhóm $O(n\log_2 n)$ là Merge sort và Heap sort. Từ thuật toán Quick sort, ta cũng có thể xây dựng được một thuật toán hiệu quả tìm phần tử trung vị (median) của một dãy số.

Người ta cũng đã chứng minh được rằng, $O(n\log_2 n)$ là ngưỡng chặn dưới của các thuật toán sắp xếp dựa trên nền tảng so sánh giá trị của các phần tử. Để vượt qua ngưỡng này, ta cần phát triển thuật toán mới theo hướng khác các thuật toán trên. Radix sort là một thuật toán như vậy. Nó được phát triển dựa trên sự mô phỏng qui trình phân phối thư của những người đưa thư. Thuật toán này đại diện cho nhóm các thuật toán sắp xếp có độ phức tạp tuyến tính. Tuy nhiên, thường thì các thuật toán này không thích hợp cho việc cài đặt trên cấu trúc dữ liệu mảng một chiều.

Chúng ta sẽ còn gặp lại các thuật toán Quick sort, Merge sort, Radix sort trong chương kế tiếp khi khảo sát các thao tác trên danh sách liên kết.

Bài tập lý thuyết

7. Xét mảng các số nguyên có nội dung như sau :

-9 -9 -5 -2 0 3 7 7 10 15

a. Tính số lần so sánh để tìm ra phần tử X = -9 bằng phương pháp:

a.1 Tìm tuyến tính
a.2 Tìm nhị phân

Nhận xét và so sánh hai phương pháp tìm nêu trên trong trường hợp này và trong trường hợp tổng quát.

b. Trong trường hợp tìm nhị phân, phần tử nào sẽ được tìm thấy (thứ 1 hay 2)

8. Xây dựng thuật toán tìm phần tử nhỏ nhất (lớn nhất) trong một mảng các số nguyên.

9. Một giải thuật sắp xếp được gọi là ổn định (stable) nếu sau khi thực hiện sắp xếp, thứ tự tương đối của các mẩu tin có khóa bằng nhau không đổi. Trong các giải thuật đã trình bày, giải thuật nào là ổn định ?

10. Trong ba phương pháp sắp xếp cơ bản (chọn trực tiếp, chèn trực tiếp, nổi bọt) phương pháp nào thực hiện sắp xếp nhanh nhất với một dãy đã có thứ tự ? Giải thích.

11. Cho một ví dụ minh hoạ ưu điểm của thuật toán Shake sort đối

với Buble sort khi sắp xếp một dãy số.

12. Xét bản cài đặt thao tác phân hoạch trong thuật toán QuickSort sau đây :

```
i  = 0;  j = n-1; x = a[n/2];
do
{
   while ( a[i]<x) i ++;
   while ( a[j]>x) j --;
   Hoanvi(a[i],a[j]);
}while (i <= j );
```

Có dãy a[0], a[1], ..., a[n-1] nào làm đoạn chương trình trên sai hay không ? Cho ví dụ minh họa.

13. Hãy xây dựng thuật toán tìm phần tử trung vị (median) của một dãy số a_1, a_2, ..., a_n dựa trên thuật toán Quick sort. Cho biết độ phức tạp của thuật toán này.

Bài tập thực hành

14. Cài đặt các thuật toán tìm kiếm và sắp xếp đã trình bày. Thể hiện trực quan các thao tác của thuật toán. Tính thời gian thực hiện của mỗi thuật toán.

15. Hãy viết hàm tìm tất cả các số nguyên tố nằm trong mảng một chiều a có n phần tử.

16. Hãy viết hàm tìm dãy con tăng dài nhất của mảng một chiều a có n phần tử (dãy con là một dãy liên tiếp các phần của a).

17. Khái niệm heap trong chương này trình bày còn được gọi là heap max vì phần tử a_1 là max của heap. Tương tự ta có thể

định nghĩa một heap min. Dùng heap min, hãy xây dựng và cài đặt thuật toán để sắp theo thứ tự giảm dãy số a có n phần tử.

18. Cài đặt thuật toán tìm phần tử trung vị (median) của một dãy số bạn đã xây dựng trong bài tập 6.

19. Hãy viết hàm đếm số đường chạy của mảng một chiều a có n phần tử (dãy con là một dãy liên tiếp các phần của a).

20. Hãy cài đặt thuật toán trộn trực tiếp mà chỉ sử dụng thêm một mảng phụ có kích thước bằng mảng cần sắp xếp A. (HD: do hai mảng con B, C tách ra từ A nên tổng số phần tử của B và C đúng bằng số phần tử của A. Hãy dùng một mảng chung Buff để lưu trữ B và C. B lưu ở đầu mảng Buff còn C lưu ở cuối – ngược từ cuối lên. Như vậy B và C sẽ không bao giờ chồng lấp lên nhau mà chỉ cần dùng 1 mảng).

21. Hãy viết hàm trộn hai mảng một chiều có thứ tự tăng b và c có m và n phần tử thành mảng một chiều a cũng có thứ tự tăng.

22. Hãy cài đặt thuật toán trộn tự nhiên. Thử viết chương trình lập bảng so sánh thời gian thực hiện của thuật toán trộn tự nhiên với thuật toán trộn trực tiếp và thuật toán quick sort bằng các thử nghiệm thực tế.

23. Hãy cài đặt thuật toán Radix sort để sắp xếp các phần tử của mảng a có n phần tử ($n \leq 100$).

24. Cài đặt thêm chức năng xuất bảng lương nhân viên theo thứ

tự tiến lương tăng dần cho bài tập 6 - chương 1.

25. *Hãy viết chương trình cho phép so sánh các thuật toán sắp xếp khác nhau bằng cách tạo ra dữ liệu thử ngẫu nhiên (về cả số phần tử trong mảng và giá trị của các phần tử) và thống kê số các thao tác so sánh, gán. Xuất kết quả ra màn hình dạng bảng.

26. *Hãy viết chương trình minh họa trực quan các thuật toán tìm kiếm và sắp xếp để hỗ trợ cho những người học môn cấu trúc dữ liệu.

CHƯƠNG 3

CẤU TRÚC DỮ LIỆU ĐỘNG

Mục tiêu

- Giới thiệu khái niệm cấu trúc dữ liệu động.
- Danh sách liên kết: tổ chức, các thuật toán, ứng dụng.

I. ĐẶT VẤN ĐỀ

Với các cấu trúc dữ liệu được xây dựng từ các kiểu cơ sở như: kiểu thực, kiểu nguyên, kiểu ký tự ... hoặc từ các cấu trúc đơn giản như mẩu tin, tập hợp, mảng ... lập trình viên có thể giải quyết hầu hết các bài toán đặt ra. Các đối tượng dữ liệu được xác định thuộc những kiểu dữ liệu này có đặc điểm chung là không thay đổi được kích thước, cấu trúc trong quá trình sống, do vậy thường cứng nhắc, gò bó khiến đôi khi khó diễn tả được thực tế vốn sinh động, phong phú. Các kiểu dữ liệu kể trên được gọi là các kiểu dữ liệu tĩnh.

Ví dụ

1. Trong thực tế, một số đối tượng có thể được định nghĩa đệ qui, ví dụ để mô tả đối tượng 'con người' cần thể hiện các thông tin tối thiểu như :
   - Họ tên
   - Số CMND

- Thông tin về cha, mẹ

Để biểu diễn một đối tượng có nhiều thành phần thông tin như trên có thể sử dụng kiểu mẩu tin. Tuy nhiên, cần lưu ý cha, mẹ của một người cũng là các đối tượng kiểu NGƯỜI, do vậy về nguyên tắc cần phải có định nghĩa như sau:

```
typedef   struct NGUOI{
   char   Hoten[30];
   intSo_CMND ;
   NGUOI  Cha,Me;
};
```

Tuy nhiên với khai báo trên, các ngôn ngữ lập trình gặp khó khăn trong việc cài đặt không vượt qua được như xác định kích thước của đối tượng kiểu NGUOI ?

2. Một số đối tượng dữ liệu trong chu kỳ sống của nó có thể thay đổi về cấu trúc, độ lớn, như danh sách các học viên trong một lớp học có thể tăng thêm, giảm đi ... Khi đó nếu cố tình dùng những cấu trúc dữ liệu tĩnh đã biết như mảng để biểu diễn những đối tượng đó, lập trình viên phải sử dụng những thao tác phức tạp, kém tự nhiên khiến chương trình trở nên khó đọc, do đó khó bảo trì và nhất là khó có thể sử dụng bộ nhớ một cách có hiệu quả.

3. Một lý do nữa làm cho các kiểu dữ liệu tĩnh không thể đáp ứng được nhu cầu của thực tế là tổng kích thước vùng nhớ dành cho tất cả các biến tĩnh chỉ là 64kb (1 Segment bộ nhớ). Khi có nhu cầu dùng nhiều bộ nhớ hơn ta phải sử dụng các ***cấu trúc dữ liệu động.***

4. Cuối cùng, do bản chất của các dữ liệu tĩnh, chúng sẽ chiếm

vùng nhớ đã dành cho chúng suốt quá trình hoạt động của chương trình. Tuy nhiên, trong thực tế, có thể xảy ra trường hợp một dữ liệu nào đó chỉ tồn tại nhất thời hay không thường xuyên trong quá trình hoạt động của chương trình. Vì vậy việc dùng các CTDL tĩnh sẽ không cho phép sử dụng hiệu quả bộ nhớ.

Do vậy, nhằm đáp ứng nhu cầu thể hiện sát thực bản chất của dữ liệu cũng như xây dựng các thao tác hiệu quả trên dữ liệu, cần phải tìm cách tổ chức kết hợp dữ liệu với những hình thức mới linh động hơn, có thể thay đổi kích thước, cấu trúc trong suốt thời gian sống. Các hình thức tổ chức dữ liệu như vậy được gọi là ***cấu trúc dữ liệu động***. Chương này sẽ giới thiệu về các cấu trúc dữ liệu động và tập trung khảo sát cấu trúc đơn giản nhất thuộc loại này là danh sách liên kết.

II. KIỂU DỮ LIỆU CON TRỎ

1. Biến không động (biến tĩnh, biến nửa tĩnh)

Khi xây dựng chương trình, lập trình viên có thể xác định được ngay những đối tượng dữ liệu luôn cần được sử dụng, không có nhu cầu thay đổi về số lượng kích thước .... do đó có thể xác định cách thức lưu trữ chúng ngay từ đầu. Các đối tượng dữ liệu này sẽ được khai báo như các biến không động. Biến không động là những biến thỏa:

- Được khai báo tường minh,
- Tồn tại khi vào phạm vi khai báo và chỉ mất khi ra khỏi phạm vi này,
- Được cấp phát vùng nhớ trong vùng dữ liệu (Data segment) hoặc là Stack (đối với biến nửa tĩnh - các biến

cục bộ).

- Kích thước không thay đổi trong suốt quá trình sống.

Do được khai báo tường minh, các biến không động có một định danh đã được kết nối với địa chỉ vùng nhớ lưu trữ biến và được truy xuất trực tiếp thông qua định danh đó.

Ví dụ :
```
int     a;      // a, b là các biến không động
char    b[10];
```

2. Kiểu con trỏ

- Cho trước kiểu **T = <V,O>**. Kiểu con trỏ - ký hiệu **"Tp"**- chỉ đến các phần tử có kiểu **"T"** được định nghĩa:

$$Tp = <Vp, Op>$$

trong đó

- **Vp** = {{các địa chỉ có thể lưu trữ những đối tượng có kiểu T}, NULL} (với NULL là một giá trị đặc biệt tượng trưng cho một giá trị không biết hoặc không quan tâm)
- **Op** = {các thao tác định địa chỉ của một đối tượng thuộc kiểu T khi biết con trỏ chỉ đến đối tượng đó} (thường gồm các thao tác tạo một con trỏ chỉ đến một đối tượng thuộc kiểu T; hủy một đối tượng dữ liệu thuộc kiểu T khi biết con trỏ chỉ đến đối tượng đó)

- Nói một cách dễ hiểu, kiểu con trỏ là kiểu cơ sở dùng lưu địa chỉ của một đối tượng dữ liệu khác.

- Biến thuộc kiểu con trỏ Tp là biến mà giá trị của nó là địa chỉ cuả một vùng nhớ ứng với một biến kiểu T, hoặc là giá

trị NULL.

! LƯU Ý

☞ Kích thước của biến con trỏ tùy thuộc vào qui ước số byte địa chỉ trong từng mô hình bộ nhớ của từng ngôn ngữ lập trình cụ thể.

Ví dụ :

- biến con trỏ trong Pascal có kích thước 4 byte (2 byte địa chỉ segment + 2 byte địa chỉ offset)

- biến con trỏ trong C có kích thước 2 hoặc 4 byte tùy vào con trỏ near (chỉ lưu địa chỉ offset) hay far (lưu cả segment lẫn offset)

☞ Cú pháp định nghĩa một kiểu con trỏ trong ngôn ngữ C :

```
typedef   <kiểu con trỏ>   *<kiểu cơ sở>;
```

Ví dụ

```
typedef     int   *intpointer;
intpointer  p;
```

hoặc

```
int*p;
```

là những khai báo hợp lệ.

- Các thao tác cơ bản trên kiểu con trỏ(minh họa bằng C)

- Khi một biến con trỏ p lưu địa chỉ của đối tượng x, ta nói '*p trỏ đến x*'.
- Gán địa chỉ của một vùng nhớ con trỏ p:

  p = <địa chỉ>;

  p = <địa chỉ> + <giá trị nguyên>;
- Truy xuất nội dung của đối tượng do p trỏ đến (*p)

3. Biến động

- Trong nhiều trường hợp, tại thời điểm biên dịch không thể xác định trước kích thước chính xác của một số đối tượng dữ liệu do sự tồn tại và tăng trưởng của chúng phụ thuộc vào ngữ cảnh của việc thực hiện chương trình. Các đối tượng dữ liệu có đặc điểm kể trên nên được khai báo như biến động. Biến động là những biến thỏa:
  - Biến không được khai báo tường minh.
  - Có thể được cấp phát hoặc giải phóng bộ nhớ khi người sử dụng yêu cầu.
  - Các biến này không theo qui tắc phạm vi (tĩnh).
  - Vùng nhớ của biến được cấp phát trong Heap.
  - Kích thước có thể thay đổi trong quá trình sống.
- Do không được khai báo tường minh nên các biến động không có một định danh được kết buộc với địa chỉ vùng nhớ cấp phát cho nó, do đó gặp khó khăn khi truy xuất đến một biến động. Để giải quyết vấn đề, biến con trỏ (là biến không động) được sử dụng để trỏ đến biến động. Khi tạo ra một biến động, phải dùng một con trỏ để lưu địa chỉ của biến này và sau đó, truy xuất đến biến động thông qua biến con

trỏ đã biết định danh.

- Hai thao tác cơ bản trên biến động là tạo và hủy một biến động do biến con trỏ 'p' trỏ đến:

Tạo ra một biến động và cho con trỏ 'p' chỉ đến nó

Hầu hết các ngôn ngữ lập trình cấp cao đều cung cấp những thủ tục cấp phát vùng nhớ cho một biến động và cho một con trỏ giữ địa chỉ vùng nhớ đó.

Một số hàm cấp phát bộ nhớ của C :

```
void* malloc(size);    // trả về con trỏ chỉ đến một vùng nhớ
                       // size byte vừa được cấp phát.
void* calloc(n,size);  // trả về con trỏ chỉ đến một vùng nhớ
                       // vừa được cấp phát gồm n phần tử,
                       // mỗi phần tử có kích thước size byte
new    // hàm cấp phát bộ nhớ trong C++
```

Hủy một biến động do p chỉ đến

Hàm **free**(*p*) huỷ vùng nhớ cấp phát bởi hàm malloc hoặc calloc do p trỏ tới

Hàm **delete** *p* huỷ vùng nhớ cấp phát bởi hàm new do p trỏ tới

Ví dụ :

```
int*    p1, p2;
// cấp phát vùng nhớ cho một biến động kiểu int
p1 = (int*)malloc(sizeof(int));
```

```
p1* = 5; // đặt giá trị năm cho biến động p1
// cấp phát biến động kiểu mảng gồm 10 phần tử kiểu int
p2 = (int*)calloc(10, sizeof(int));
(p2+3)* = 0; // đặt giá trị 0 cho phần tử thứ 4 của mảng p2
free(p1); free(p2);
```

III. DANH SÁCH LIÊN KẾT (LINK LIST)

1. Định nghĩa

Cho T là một kiểu được định nghiã trước, kiểu danh sách Tx gồm các phần tử thuộc kiểu T được định nghĩa là:

$$Tx = <Vx, Ox>$$

trong đó:

Vx = {tập hợp có thứ tự các phần tử kiểu T được móc nối với nhau theo trình tự tuyến tính};

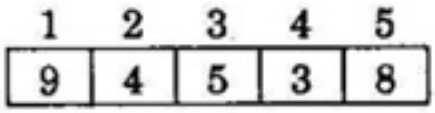
Ox = {Tạo danh sách; Tìm một phần tử trong danh sách; Chèn một phần tử vào danh sách; Huỷ một phần tử khỏi danh sách ; Liệt kê danh sách, Sắp xếp danh sách ...}

Ví du: Hồ sơ các học sinh của một trường được tổ chức thành danh sách gồm nhiều hồ sơ của từng học sinh; số lượng học sinh trong trường có thể thay đổi do vậy cần có các thao tác thêm, hủy một hồ sơ; để phục vụ công tác giáo vụ cần thực hiện các thao tác tìm hồ sơ của một học sinh, in danh sách hồ sơ ...

2. Các hình thức tổ chức danh sách

Có nhiều hình thức tổ chức mối liên hệ tuần tự giữa các phần tử trong cùng một danh sách:

- Mối liên hệ giữa các phần tử được thể hiện ngầm: mỗi phần tử trong danh sách được đặc trưng bằng chỉ số. Cặp phần tử x_i, x_{i+1} được xác định là kế cận trong danh sách nhờ vào quan hệ giữa cặp chỉ số i và (i+1). Với hình thức tổ chức này, các phần tử của danh sách thường bắt buộc phải lưu trữ liên tiếp trong bộ nhớ để có thể xây dựng công thức xác định địa chỉ phần tử thứ i:



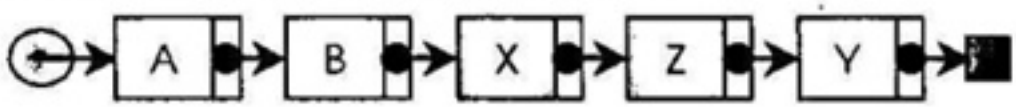
address(i) = address(1) + (i-1)*sizeof(T)

Có thể xem mảng và tập tin là những danh sách đặc biệt được tổ chức theo hình thức liên kết "ngầm" giữa các phần tử. Tuy nhiên mảng có một đặc trưng giới hạn là số phần tử mảng cố định, do vậy không có thao tác thêm, hủy trên mảng; trường hợp tập tin thì các phần tử được lưu trữ trên bộ nhớ phụ có những đặc tính lưu trữ riêng sẽ được trình bày chi tiết ở giáo trình Cấu trúc dữ liệu 2.

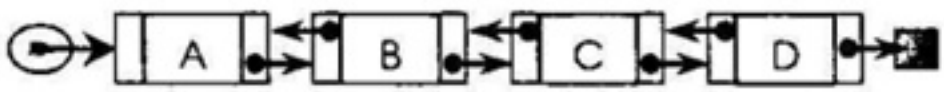
Cách biểu diễn này cho phép truy xuất ngẫu nhiên, đơn giản và nhanh chóng đến một phần tử bất kỳ trong danh sách, nhưng lại hạn chế về mặt sử dụng bộ nhớ. Đối với mảng, số phần tử được xác định trong thời gian biên dịch và cần cấp phát vùng nhớ liên tục. Trong trường hợp tổng kích thước bộ nhớ trống còn đủ để chứa toàn bộ mảng nhưng các ô nhớ trống lại không nằm kế cận nhau thì cũng không cấp phát vùng nhớ cho mảng được. Ngoài ra do kích thước mảng cố định mà số phần tử của danh sách lại khó dự trù chính xác nên có thể gây ra tình trạng thiếu hụt hay lãng phí bộ nhớ. Hơn nữa các thao tác thêm, hủy một phần tử vào danh sách được thực hiện không tự nhiên trong hình thức tổ chức này.

- Mối liên hệ giữa các phần tử được thể hiện tường minh: mỗi phần tử ngoài các thông tin về bản thân còn chứa một liên kết (địa chỉ) đến phần tử kế trong danh sách nên còn được gọi là danh sách móc nối. Do liên kết tường minh, với hình thức này các phần tử trong danh sách không cần phải lưu trữ kế cận trong bộ nhớ nên khắc phục được các khuyết điểm của hình thức tổ chức mảng, nhưng việc truy xuất đến một phần tử đòi hỏi phải thực hiện truy xuất qua một số phần tử khác. Có nhiều kiểu tổ chức liên kết giữa các phần tử trong danh sách như :

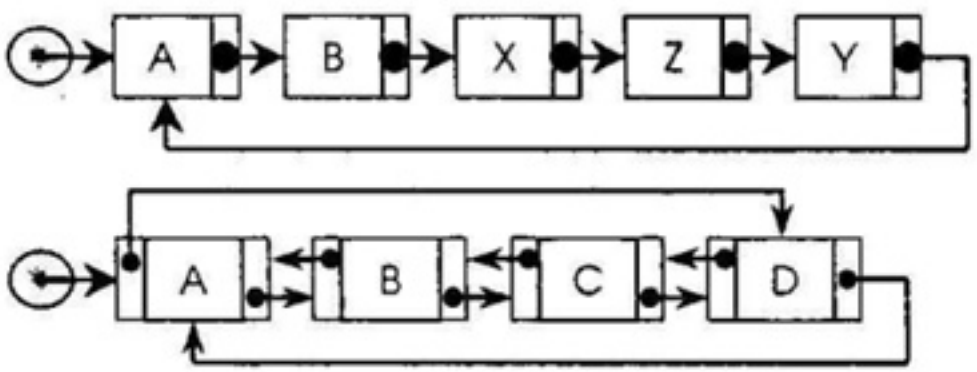
  - **Danh sách liên kết đơn:** mỗi phần tử liên kết với phần tử đứng sau nó trong danh sách:



  - **Danh sách liên kết kép:** mỗi phần tử liên kết với các phần tử đứng trước và sau nó trong danh sách:



  - **Danh sách liên kết vòng**: phần tử cuối danh sách liên kết với phần tử đầu danh sách:



Hình thức liên kết này cho phép các thao tác thêm, hủy trên danh sách được thực hiện dễ dàng, phản ánh được bản chất linh động của danh sách.

Nhằm giới thiệu cấu trúc dữ liệu động, chương này sẽ trình bày các danh sách với hình thức tổ chức liên kết tường minh.

IV. DANH SÁCH ĐƠN (XÂU ĐƠN)

1. Tổ chức danh sách đơn theo cách cấp phát liên kết

Cấu trúc dữ liệu của một phần tử trong danh sách đơn

Mỗi phần tử của danh sách đơn là một cấu trúc chứa hai thông tin :

- *Thành phần dữ liệu:* lưu trữ các thông tin về bản thân phần tử .
- *Thành phần mối liên kết:* lưu trữ địa chỉ của phần tử kế tiếp trong danh sách, hoặc lưu trữ giá trị NULL nếu là phần tử cuối danh sách.

Ta có định nghĩa tổng quát

```
typedef  struct  tagNode
{
   Data  Info;          // Data là kiểu đã định nghĩa trước
   struct tagNode* pNext; // con trỏ chỉ đến cấu trúc node
}NODE;
```

Ví dụ : Định nghĩa danh sách đơn lưu trữ hồ sơ sinh viên:

```
typedef  struct  SinhVien
{  char  Ten[30];
   int   MaSV;
}SV;
```

```
typedef  struct  SinhvienNode
{  SV Info;
   struct SinhvienNode* pNext;
}SVNode;
```

- Một phần tử trong danh sách đơn là một biến động sẽ được yêu cầu cấp phát khi cần. Và danh sách đơn chính là sự liên kết các biến động này với nhau, do vậy đạt được sự linh động khi thay đổi số lượng các phần tử

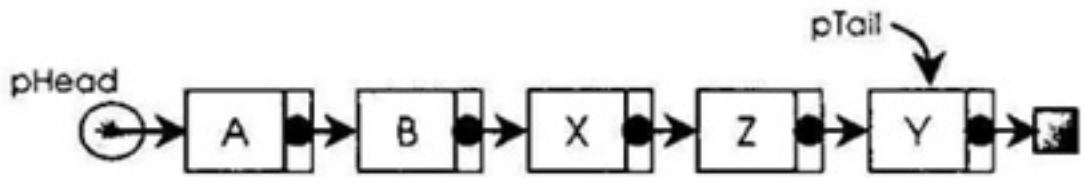
- Nếu biết được địa chỉ của phần tử đầu tiên trong danh sách đơn thì có thể dựa vào thông tin pNext của nó để truy xuất đến phần tử thứ 2 trong xâu, và lại dựa vào thông tin Next của phần tử thứ 2 để truy xuất đến phần tử thứ 3... Nghĩa là để quản lý một xâu đơn chỉ cần biết địa chỉ phần tử đầu xâu. Thường một con trỏ Head sẽ được dùng để lưu trữ địa chỉ phần tử đầu xâu, ta gọi Head là đầu xâu. Ta có khai báo:

```
NODE   *pHead;
```

- Tuy về nguyên tắc chỉ cần quản lý xâu thông qua đầu xâu pHead, nhưng thực tế có nhiều trường hợp cần làm việc với phần tử cuối xâu, khi đó mỗi lần muốn xác định phần tử cuối xâu lại phải duyệt từ đầu xâu. Để tiện lợi, có thể sử dụng thêm một con trỏ pTail giữ địa chỉ phần tử cuối xâu. Khai báo pTail như sau :

```
NODE   *pTail;
```

Lúc này có xâu đơn:



Ta sẽ quản lý xâu đơn theo phương thức và nếu trong giới hạn của giáo trình này.

2. Các thao tác cơ bản trên danh sách đơn

Giả sử có các định nghĩa:

```
typedef  struct  tagNode
{
   Data  Info;
   struct tagNode* pNext;
}NODE;          // kiểu của một phần tử trong danh sách

typedef  struct  tagList
{
   NODE* pHead;
   NODE* pTail;
}LIST;          // kiểu danh sách liên kết
NODE   *new_ele // giữ địa chỉ của một phần tử mới được tạo
Data   x; // lưu thông tin về một phần tử sẽ được tạo
```

và đã xây dựng thủ tục GetNode để tạo ra một phần tử cho danh sách với thông tin chứa trong x:

```
NODE* GetNode(Data x)
{  NODE *p;
   // Cấp phát vùng nhớ cho phần tử
   p = new NODE;
   if ( p==NULL)    {
      printf("Không đủ bộ nhớ"); exit(1);
   }
   p ->Info = x; // Gán thông tin cho phần tử p
   p->pNext = NULL;
   return p;
}
```

Phần tử do new_ele giữ địa chỉ tạo bởi câu lệnh :

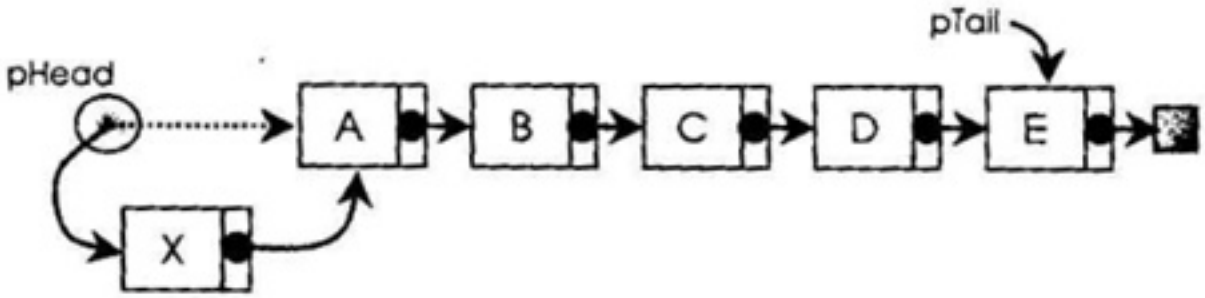
```
new_ele = GetNode(x);
```

được gọi là new_ele.

Chèn một phần tử vào danh sách

Có ba loại thao tác chèn new_ele vào xâu:

Cách 1: Chèn vào đầu danh sách



- **Thuật toán**

Bắt đầu

Nếu Danh sách rỗng Thì

B1.1 : Head = new_elelment;

B1.2 : Tail = Head;

Ngược lại

B2.1 : new_ele ->pNext = Head;

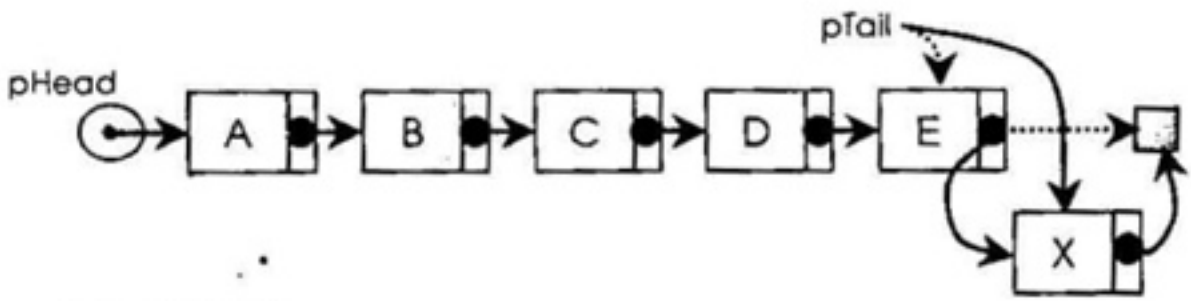
B2.2 : Head = new_ele ;

- **Cài đặt**

```
void AddFirst(LIST &l, NODE* new_ele)
{
   if (l.pHead==NULL)   //Xâu rỗng
   {
      l.pHead = new_ele; l.pTail = l.pHead;
   }
   else
   {
```

```
        new_ele->pNext = l.pHead;
        l.pHead = new_ele;
    }
}
NODE* InsertHead(LIST &l, Data x)
{
    NODE* new_ele = GetNode(x);

    if (new_ele ==NULL)  return NULL;
    if (l.pHead==NULL)
    {
        l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {
        new_ele->pNext = l.pHead;
        l.pHead = new_ele;
    }
    return new_ele;
}
```

Cách 2: Chèn vào cuối danh sách



- **Thuật toán**

Bắt đầu

Nếu Danh sách rỗng Thì

B1.1 : Head = new_elelment;

B1.2 : Tail = Head;

Ngược lại

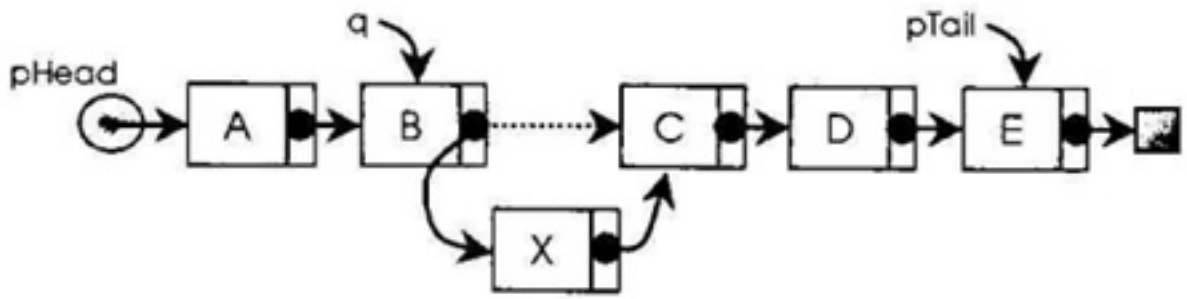
B2.1 : Tail ->pNext = new_ele;

B2.2 : Tail = new_ele ;

- **Cài đặt**

```
void AddTail(LIST &l, NODE *new_ele)
{
   if (l.pHead==NULL)
   {
      l.pHead = new_ele; l.pTail = l.pHead;
   }
   else
   {
      l.pTail->Next = new_ele;
      l.pTail = new_ele;
   }
}

NODE* InsertTail(LIST &l, Data x)
{
   NODE* new_ele = GetNode(x);

   if (new_ele ==NULL)  return NULL;
   if (l.pHead==NULL)
   {
      l.pHead = new_ele; l.pTail = l.pHead;
   }
   else
   {
      l.pTail->Next = new_ele;
      l.pTail = new_ele;
   }
   return new_ele;
}
```

Cách 3 : Chèn vào danh sách sau một phần tử q



- **Thuật toán**

Bắt đầu

Nếu (q != NULL) thì

B1 : new_ele -> pNext = q->pNext;

B2 : q->pNext = new_ele ;

- **Cài đặt**

```
void AddAfter(LIST &l,NODE *q, NODE* new_ele)
{
   if ( q!=NULL)
   {
      new_ele->pNext = q->pNext;
      q->pNext = new_ele;
      if(q == l.pTail)
         l.pTail = new_ele;
   }
   else //chèn vào đầu danh sách
      AddFirst(l, new_ele);
}

void InsertAfter(LIST &l,NODE *q, Data x)
{
   NODE* new_ele = GetNode(x);

   if (new_ele ==NULL)  return NULL;

   if ( q!=NULL)
   {
      new_ele->pNext = q->pNext;
      q->pNext = new_ele;
      if(q == l.pTail)
         l.pTail = new_ele;
   }
   else //chèn vào đầu danh sách
      AddFirst(l, new_ele);
}
```

Tìm một phần tử trong danh sách đơn

- **Thuật toán**

Xâu đơn đòi hỏi truy xuất tuần tự, do đó chỉ có thể áp dụng thuật toán tìm tuyến tính để xác định phần tử trong xâu có khoá **k**. Sử dụng một con trỏ phụ trợ **p** để lần lượt trỏ đến các phần tử trong xâu. Thuật toán được thể hiện như sau :

Bước 1

p = Head; //Cho p trỏ đến phần tử đầu danh sách

Bước 2

Trong khi (p != NULL) và (p->pNext != k) thực hiện:

B21 : p:=p->Next;// Cho p trỏ tới phần tử kế

Bước 3

Nếu p != NULL thì p trỏ tới phần tử cần tìm

Ngược lại: không có phần tử cần tìm.

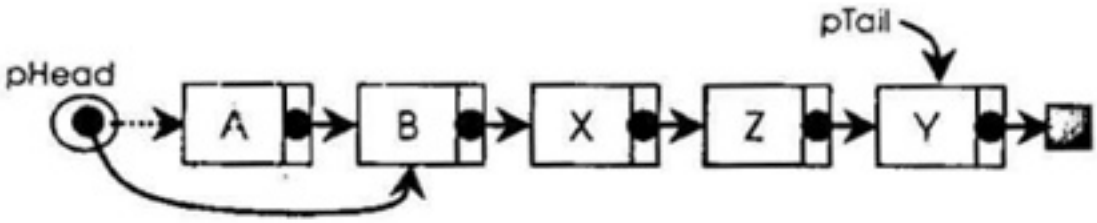
- **Cài đặt**

```
NODE *Search(LIST l, Data  k)
{  NODE  *p;
   p = l.pHead;
   while((p!= NULL)&&(p->Info != x))
      p = p->pNext;
   return p;
}
```

Hủy một phần tử khỏi danh sách

Có ba loại thao tác thông dụng hủy một phần tử ra khỏi xâu. Chúng ta sẽ lần lượt khảo sát chúng. Lưu ý là khi cấp phát bộ nhớ, chúng ta đã dùng hàm new. Vì vậy khi giải phóng bộ nhớ ta phải dùng hàm delete.

Hủy phần tử đầu xâu



- **Thuật toán**

Bắt đầu

Nếu (Head != NULL) thì

B1: p = Head; // p là phần tử cần hủy

B2:

B21 : Head = Head->pNext; // tách p ra khỏi xâu

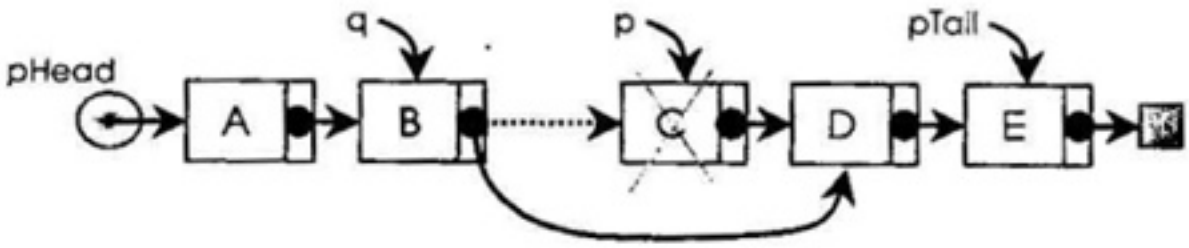
B22 : free(p); // Hủy biến động do p trỏ đến

B3: Nếu Head=NULL thì Tail = NULL; //Xâu rỗng

- **Cài đặt**

```
Data RemoveHead(LIST &l)
{  NODE  *p;
   Data  x = NULLDATA;

   if ( l.pHead != NULL)
   {
      p = l.pHead; x = p->Info;
      l.pHead = l.pHead->pNext;
      delete p;
      if(l.pHead == NULL)  l.pTail = NULL;
   }
   return x;
    }
```

Hủy một phần tử đứng sau phần tử q



- ***Thuật toán***

Bắt đầu

Nếu (q!= NULL) thì

B1: p = q->Next; // p là phần tử cần hủy

B2: Nếu (p != NULL) thì // q không phải là cuối xâu

B2.1 : q->Next = p->Next; // tách p ra khỏi xâu

B2.2 : free(p); // Hủy biến động do p trỏ đến

- **Cài đặt**

```
void RemoveAfter (LIST &l, NODE *q)
{  NODE  *p;

   if ( q != NULL)
   {
      p = q ->pNext ;
      if ( p != NULL)
      {
         if(p == l.pTail)  l.pTail = q;
         q->pNext = p->pNext;
         delete p;
      }
   }
   else
      RemoveHead(l);
}
```

Hủy một phần tử có khoá k

- **Thuật toán**

Bước 1

Tìm phần tử p có khóa k và phần tử q đứng trước nó

Bước 2

Nếu (p!= NULL) thì // tìm thấy k

Hủy p ra khỏi xâu tương tự hủy phần tử sau q;

Ngược lại

Báo không có k;

- **Cài đặt**

```
int RemoveNode(LIST &l, Data k)
{  NODE  *p = l.pHead;
   NODE  *q = NULL;

   while( p != NULL)
   {
      if(p->Info == k) break;
      q = p; p = p->pNext;
   }
   if(p == NULL) return 0; //Không tìm thấy k
   if(q != NULL)
   {
      if(p == l.pTail)
      l.pTail = q;
      q->pNext = p->pNext;
      delete p;
   }
   else //p là phần tử đầu xâu
   {
      l.pHead = p->pNext;
      if(l.pHead == NULL)
      l.pTail = NULL;
   }
   return 1;
}
```

Duyệt danh sách là thao tác thường được thực hiện khi có nhu cầu xử lý các phần tử của danh sách theo cùng một cách thức hoặc khi cần lấy thông tin tổng hợp từ các phần tử của danh sách như:

- Đếm các phần tử của danh sách,
- Tìm tất cả các phần tử thoả điều kiện,
- Huỷ toàn bộ danh sách (và giải phóng bộ nhớ)

Để duyệt danh sách (và xử lý từng phần tử) ta thực hiện các thao tác sau:

- **Thuật toán**

Bước 1

p = Head; //Cho p trỏ đến phần tử đầu danh sách

Bước 2

Trong khi (Danh sách chưa hết) thực hiện
B21 : Xử lý phần tử p;
B22 : p:=p->pNext; // Cho p trỏ tới phần tử kế

- **Cài đặt**

```
void ProcessList (LIST &l)
{  NODE  *p;

   p = l.pHead;
   while (p!= NULL)
   {

      ProcessNode(p); // xử lý cụ thể tùy ứng dụng
      p = p->pNext;
   }
}
```

! LƯU Ý

☞ Để huỷ toàn bộ danh sách, ta có một chút thay đổi trong thủ tục duyệt (xử lý) danh sách trên (ở đây, thao tác xử lý bao gồm hành động giải phóng một phần tử, do vậy phải cập nhật các liên kết liên quan).

- **Thuật toán**

Bước 1

Trong khi (Danh sách chưa hết) thực hiện

B1.1:

p = Head;

Head:=Head->pNext; // Cho p trỏ tới phần tử kế

B1.2:

Hủy p;

Bước 2

Tail = NULL; //Bảo đảm tính nhất quán khi xâu rỗng

- **Cài đặt**

```
void ReamoveList(LIST &l)
{  NODE  *p;

   while (l.pHead!= NULL)
   {
      p = l.pHead;

      l.pHead = p->pNext;
      delete p;
   }
   l.pTail = NULL;
}
```

3. Sắp xếp danh sách

Các cách tiếp cận

Một danh sách có thứ tự (danh sách được sắp) là một danh sách mà các phần tử của nó được sắp xếp theo một thứ tự nào đó dựa trên một trường khoá. Ví dụ, danh sách các phần tử số có thứ tự tăng là danh sách mà với mọi cặp phần tử X, Y ta luôn có $X \leq Y$ nếu X xuất hiện trước Y trong danh sách (danh sách có một hoặc không có phần tử nào được xem là một danh sách được sắp). Để sắp xếp một danh sách, ta có thể thực hiện một trong hai phương án sau:

- *Phương án 1*: Hoán vị nội dung các phần tử trong danh sách (thao tác trên vùng Info). Với phương án này, có thể chọn một trong những thuật toán sắp xếp đã biết để cài đặt lại trên xâu như thực hiện trên mảng, điểm khác biệt duy nhất là cách thức truy xuất đến các phần tử trên xâu thông qua liên kết thay vì chỉ số như trên mảng. Do dựa trên việc hoán vị nội dung của các phần tử, phương pháp này đòi hỏi sử dụng thêm vùng nhớ trung gian nên chỉ thích hợp với các xâu có các phần tử có thành phần Info kích thước nhỏ. Hơn nữa số lần hoán vị có thể lên đến bậc n^2 với xâu n phần tử. Khi kích thước của trường Info lớn, việc hoán vị giá trị của hai phân tử sẽ chiếm chi phí đáng kể. Điều này sẽ làm cho thao tác xấp xếp chậm lại. Như vậy, phương án này không tận dụng được các ưu điểm của xâu .

Ví dụ

Cài đặt thuật toán sắp xếp chọn trực tiếp trên xâu :

```
void  ListSelectionSort (LIST &l)
{   NODE   *min; // chỉ đến phần tử có giá trị nhỏ nhất trong xâu
    NODE   *p,*q;
```

```
    p = l.pHead;
    while(p != l.pTail)
    {
        q = p->pNext; min = p;
        while(q != NULL)
        {
            if(q->Info< min->Info )
                min = q; // ghi nhận vị trí phần tử min hiện hành
            q = q->pNext;
        }
        // Hoán vị nội dung 2 phần tử
        Hoanvi(min->Info, p->Info]);
        p = p->pNext;
    }
}
```

- *Phương án 2*: Thay đổi các mối liên kết (thao tác trên vùng Next)

Do các nhược điểm của các phương pháp sắp xếp theo phương án 1, khi dữ liệu lưu tại mỗi phần tử trong xâu có kích thước lớn người ta thường dùng một cách tiếp cận khác. Thay vì hoán đổi giá trị, ta sẽ tìm cách thay đổi trình tự móc nối của các phần tử sao cho tạo lập nên được thứ tự mong muốn. Cách tiếp cận này sẽ cho phép ta chỉ thao tác trên các móc nối (trường pNext). Như ta đã biết, kích thước của trường này không phụ thuộc vào bản chất dữ liệu lưu trong xâu vì có bằng đúng một con trỏ (2 byte hoặc 4 byte trong môi trường 16 bit và 4 byte hoặc 8 byte trong môi trường 32 bit, ...). Tuy nhiên thao tác trên các móc nối thường sẽ phức tạp hơn là thao tác trực tiếp trên dữ liệu. Vì vậy, ta cần cân nhắc kỹ lưỡng trước khi chọn cách tiếp cận. Nếu dữ liệu không quá lớn thì ta nên chọn phương án 1 hoặc một thuật toán hiệu quả nào đó.

Một trong những cách thay đổi móc nối đơn giản nhất là tạo một danh sách mới là danh sách có thứ tự từ danh sách cũ (đồng thời hủy danh sách cũ). Giả sử danh sách mới sẽ được

quản lý bằng con trỏ đầu xâu Result, ta có phương án 2 của thuật toán chọn trực tiếp như sau :

Bước 1: Khởi tạo danh sách mới Result là rỗng;
Bước 2: Tìm trong danh sách cũ một phần tử nhỏ nhất;
Bước 3: Tách min khỏi danh sách l;
Bước 4: Chèn min vào cuối danh sách Result;
Bước 5: Lặp lại bước 2 khi chưa hết danh sách Head;

Ta có thể cài đặt thuật toán trên như sau:

```
void  ListSelectionSort2 (LIST &l)
{  LIST  lRes;
   NODE  *min; // chỉ đến phần tử có giá trị nhỏ nhất trong xâu
   NODE  *p,*q, minprev;
   lRes.pHead = lRes.pTail = NULL; // khởi tạo lRes
   while(l.pHead != NULL)
   {
      p = l.pHead;
      q = p->pNext; min = p; minprev = NULL;
      while(q != NULL)
      {
         if(q->Info< min->Info ) {
            min = q; minprev = p
         }
         p = q; q = q->pNext;
      }
      if(minprev != NULL)
         minprev->pNext = min->pNext;
      else
         l.pHead = min->pNext;
      min->pNext = NULL;
      AddTail(lRes, min);
   }
   l = lRes;
}
```

Thuật toán Quick sort

Trong số các thuật toán sắp xếp, có lẽ nổi tiếng nhất về hiệu quả là thuật toán Quick sort. Các cài đặt của thuật toán này thường thấy trên cấu trúc dữ liệu mảng. Trong chương 2 chúng ta đã khảo sát thuật toán này. Tuy nhiên ít ai để ý rằng nó cũng là một trong những thuật toán sắp xếp hiệu quả nhất trên xâu. Hơn nữa, khi cài đặt trên xâu, bản chất của thuật toán này thể hiện một cách rõ ràng hơn bao giờ hết.

- **Thuật toán Quick sort**

Bước 1

Chọn X là phần tử đầu xâu L làm phần tử cắm canh. Loại X ra khỏi L.

Bước 2

Tách xâu L ra làm hai xâu L_1 (gồm các phần tử nhỏ hơn hay bằng X) và L_2 (gồm các phần tử lớn hơn X).

Bước 3

Nếu L_1 != NULL thì Quick sort (L_1).

Bước 4

Nếu L_2 != NULL thì Quick sort (L_2).

Bước 5

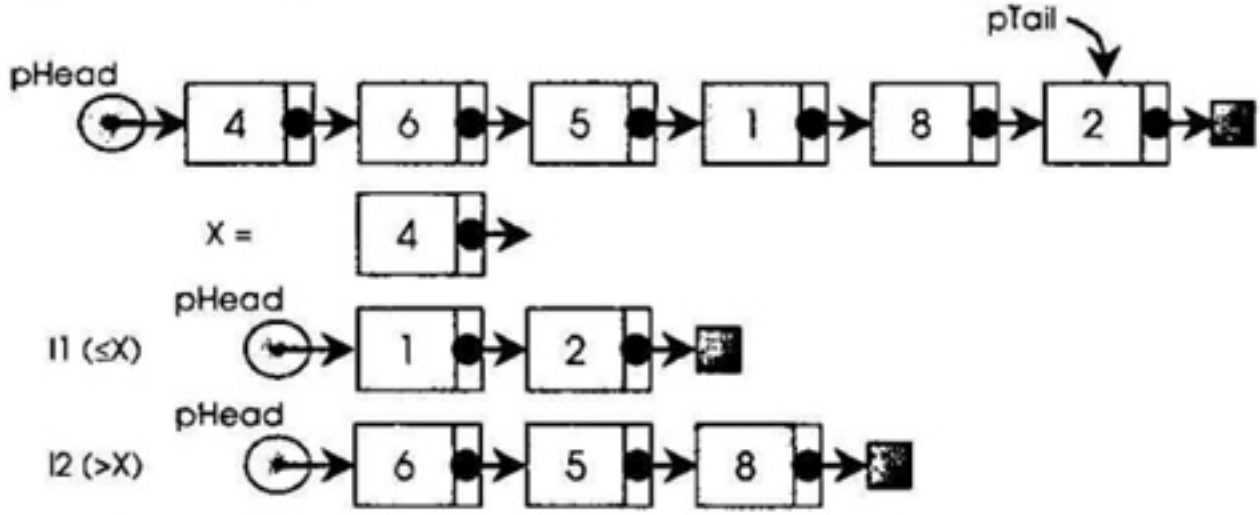
Nối L1, X, và L2 lại theo trình tự ta có xâu L đã được sắp xếp.

- **Ví dụ**

Cho dãy số a: 4 6 5 1 8 2

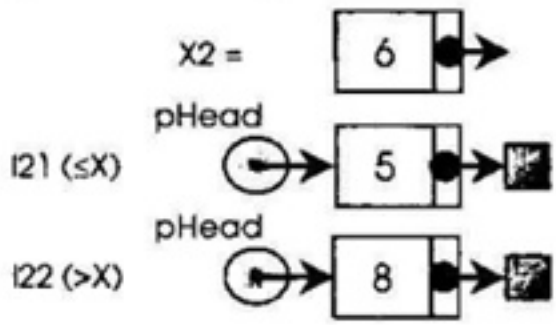
Sắp xếp l1

Chọn X = 4 làm phần tử cắm canh và tách l thành l1, l2:

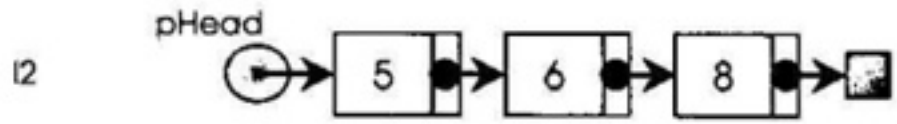


Sắp xếp l2

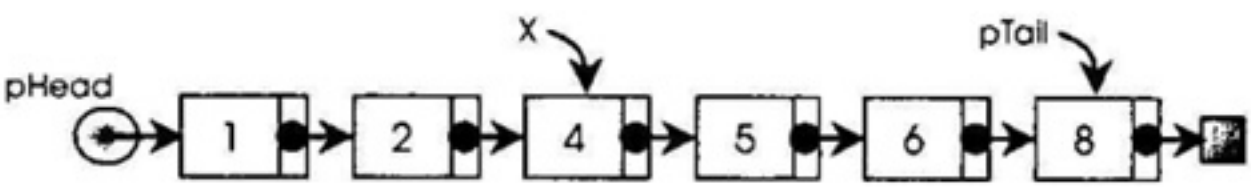
Chọn X = 6 làm phần tử cắm canh và tách l2 thành l21, l22:



Nối l12, X2, l22 thành l2:



Nối l1, X, l2 thành l:



- **Cài đặt**

```
void  ListQSort(LIST & l)
{  NODE  *p, *X;   // X chỉ đến phần tử cắm canh
   LIST  l1, l2;

   if(l.pHead == l.pTail) return;//đã có thứ tự
   l1.pHead == l1.pTail = NULL; //khởi tạo
   l2.pHead == l2.pTail = NULL;
   X = l.pHead; l.pHead = X->pNext;
   while(l.pHead != NULL) //Tách l thành l1, l2;
   {
      p = l.pHead;
      l.pHead = p->pNext; p->pNext = NULL;
      if (p->Info <= X->Info)
         AddTail(l1, p);
      else
         AddTail(l2, p);
   }
   ListQSort(l1); //Gọi đệ qui để sort l1
   ListQSort(l2); //Gọi đệ qui để sort l2
   //Nối l1, X và l2 lại thành l đã sắp xếp.
   if(l1.pHead != NULL)
   {
      l.pHead = l1.pHead; l1.pTail->pNext = X;
   }
   else
      l.pHead = X;
   X->pNext = l2;
   if(l2.pHead != NULL)
      l.pTail = l2.pTail;
   else
      l.pTail = X;
}
```

Như chúng ta đã thấy, Quick sort trên xâu đơn đơn giản hơn phiên bản của nó trên mảng một chiều nhiều. Hãy cài đặt thử thuật toán này các bạn sẽ thấy hiệu quả của nó khó có thuật toán nào sánh bằng. Một điều đáng lưu ý là khí dùng Quick sort sắp xếp một xâu đơn, ta chỉ có một chọn lựa phần tử cắm canh duy nhất hợp lý là phần tử đầu xâu. Chọn bất kỳ phần tử nào khác cũng làm tăng chi phí một cách không cần thiết do cấu trúc tự nhiên của xâu.

Thuật toán Merge sort

Cũng như thuật toán Quick sort, Merge sort là một trong những thuật toán sắp xếp hiệu quả nhất trên xâu. Cài đặt của thuật toán này trên cấu trúc dữ liệu mảng rất rắc rối như các bạn đã thấy trong chương 2. Người ta hay nhắc đến Merge sort như là một thuật toán sắp xếp trên file (sắp xếp ngoài). Cũng như Quick sort, khi cài đặt trên xâu, bản chất của thuật toán này thể hiện rất rõ ràng.

- **Thuật toán Merge sort**

Bước 1

Phân phối luân phiên từng đường chạy của xâu L vào 2 xâu con L_1 và L_2.

Bước 2

Nếu L_1 != NULL thì Merge sort (L_1).

Bước 3

Nếu L_2 != NULL thì Merge sort (L_2).

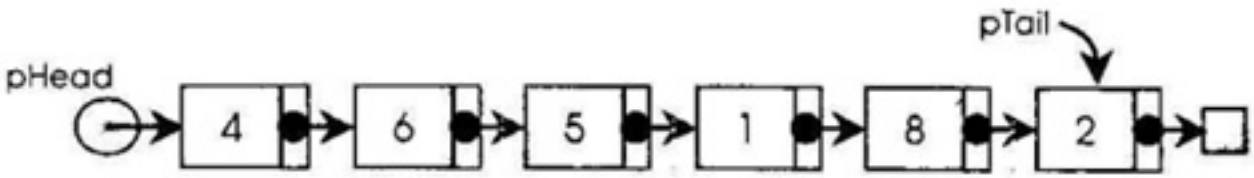
Bước 4

Trộn L_1 và L_2 đã sắp xếp lại ta có xâu L đã được sắp xếp.

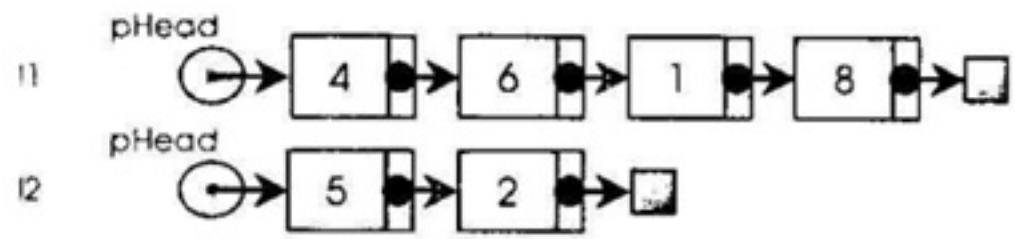
- **Ví dụ**

Cho dãy số a:

4 6 5 1 8 2

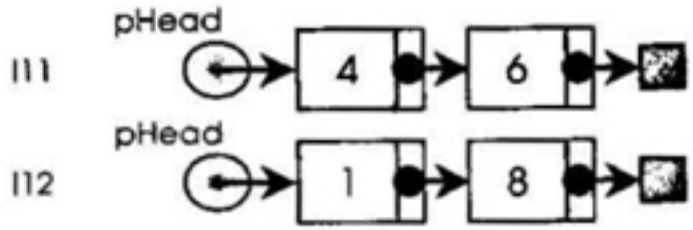


Phân phối các đường chạy của l vào l1, l2:

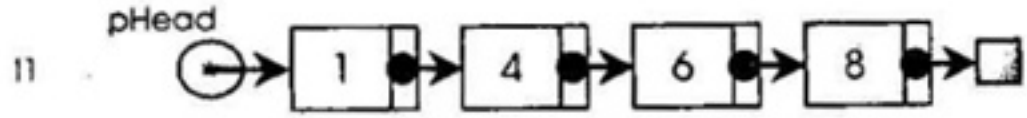


Sắp xếp l1

Phân phối các đường chạy của l1 vào l11, l12:

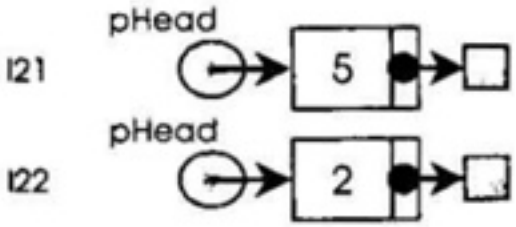


Trộn l11, l12 lại thành l1:

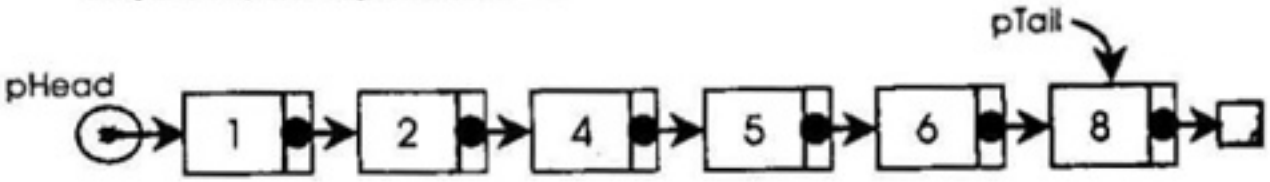


Sắp xếp l2

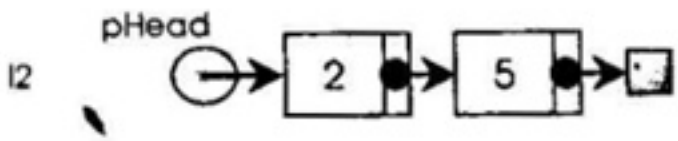
Phân phối các đường chạy của l2 vào l21, l22:



Trộn l11, l12 lại thành l1:



Trộn l1, l2 lại thành l:



- **Cài đặt**

```
void  ListMergeSort(LIST & l)
{  LIST  l1, l2;

   if(l.pHead == l.pTail) return;//đã có thứ tự
   l1.pHead == l1.pTail = NULL; //khởi tạo
   l2.pHead == l2.pTail = NULL;
```

```
    //Phân phối l thành l1 và l2 theo từng đường chạy
    DistributeList(l, l1, l2);
    ListMergeSort(l1); //Gọi đệ qui để sort l1
    ListMergeSort(l2); //Gọi đệ qui để sort l2
    //Trộn l1 và l2 đã có thứ tự thành l
    MergeList(l, l1, l2);
}
```

Trong đó, các hàm DistributeList và MergeList được viết như sau:

```
void  DistributeList(LIST& l,LIST& l1,LIST& l2)
{  NODE  *p;

   do //Tách l thành l1, l2;
   {
      p = l.pHead;
      l.pHead = p->pNext; p->pNext = NULL;


      AddTail(l1, p);
   }while((l.pHead)&&(p->Info<=l.pHead->Info));
   if(l.pHead)
      DistributeList(l, l2, l1);
   else
      l.pTail = NULL;
}

void  MergeList(LIST& l,LIST& l1,LIST& l2)
{  NODE  *p;

   while((l1.pHead)&&(l2.pHead))
   {
      if(l1.pHead->Info <= l2.pHead->Info){
         p = l1.pHead;
         l1.pHead = p->pNext;
      }
      else  {
         p = l2.pHead;
         l2.pHead = p->pNext;
      }
      p->pNext = NULL; AddTail(l, p);
```

```
    };
    if(l1.pHead) {//Nối phần còn lại của l1 vào cuối l
        l.pTail->pNext = l1.pHead;
        l.pTail = l1.pTail;
    }
    else if(l2.pHead) {//Nối phần còn lại của l2 vào cuối l
        l.pTail->pNext = l2.pHead;
        l.pTail = l2.pTail;
    }
}
```

Như chúng ta đã thấy, Merge sort trên xâu đơn đơn giản hơn phiên bản của nó trên mảng một chiều. Một điều đáng lưu ý là khi dùng Merge sort sắp xếp một xâu đơn, ta không cần dùng thêm vùng nhớ phụ như khi cài đặt trên mảng một chiều. Ngoài ra, thủ tục Merge trên xâu cũng không phức tạp như trên mảng vì ta chỉ phải trộng hai xâu đã có thứ tự, trong khi trên mảng ta phải trộn hai mảng bất kỳ.

Thuật toán Radix sort

Thuật toán Radix sort đã được giới thiệu trong chương 2. Khi cài đặt trên cấu trúc dữ liệu mảng một chiều, thuật toán này gặp một hạn chế lớn là đòi hỏi thêm quá nhiều bộ nhớ. Trong chương 2, chúng ta cũng đã đề cập đến khả năng cài đặt trên danh sách liên kết của thuật toán này. Sau đây là chi tiết thuật toán:

- **Thuật toán Radix sort**

Bước 1

Khởi tạo các danh sách (lô) rỗng B_0, B_1, ..., B_9; k = 0;

Bước 2

Trong khi L khác rỗng:

B21: p = L.pHead; L.pHead->pNext;

B22: Đặt phần tử p vào cuối lô B_d với d là chữ số thứ k của L.pHead->Info;

Bước 3

Nối B_0, B_1, ..., B_9 lại thành L; Làm B_0, B_1, ..., B_9;

Bước 4

k = k+1;

Nếu k < m: quay lại Bước 2

- **Cài đặt**

```
void  ListRadixSort(LIST & l, int m)
{  LIST  B[10];
   NODE  *p;
   inti, k;
   if(l.pHead == l.pTail) return;//đã có thứ tự
   for(i = 0; i < 10; i++)
   B[i].pHead = B[i].pTail = NULL;
   for(k = 0; k < m; k++)
   {
      while(l.pHead) {
         p = l.pHead;
         l.pHead = p->pNext; p->pNext = NULL;
         i = GetDigit(p->Info, k);
         AddTail(B[i], p);
      }
      l = B[0];
      for(i = 1; i < 10; i++)
         AppendList(l, B[i]);//Nối B[i] vào cuối l
```

Trong đó, các hàm AppendList và GetDigit được viết như sau:

```
void  AppendList(LIST& l,LIST& l1)
{
   if(l.pHead) {
      l.pTail->pNext = l1.pHead;
```

```
            l.pTail = l1. pTail;
        }
        else //xâu l rỗng
            l = l1;
    }

    int GetDigit(unsign long N, int k)
    {
        switch(k){
            case 0:  return (N % 10);
            case 1:  return ((N/10) % 10);
            case 2:  return ((N/100) % 10);
            case 3:  return ((N/1000) % 10);
            case 4:  return ((N/10000) % 10);
            case 5:  return ((N/100000) % 10);
            case 6:  return ((N/1000000) % 10);
            case 7:  return ((N/10000000) % 10);
            case 8:  return ((N/100000000) % 10);
            case 9:  return ((N/1000000000) % 10);
        }
    }
```

4. Các cấu trúc đặc biệt của danh sách đơn

Stack

Stack là một vật chứa (container) các đối tượng làm việc theo cơ chế **LIFO** (**L**ast **I**n **F**irst **O**ut) nghĩa là việc thêm một đối tượng vào stack hoặc lấy một đối tượng ra khỏi stack được thực hiện theo cơ chế "Vào sau ra trước".

Các đối tượng có thể được thêm vào stack bất kỳ lúc nào nhưng chỉ có đối tượng thêm vào sau cùng mới được phép lấy ra khỏi stack.

Thao tác thêm một đối tượng vào stack thường được gọi là

"Push". Thao tác lấy một đối tượng ra khỏi stack gọi là "Pop".

Trong tin học, CTDL stack có nhiều ứng dụng: khử đệ qui, tổ chức lưu vết các quá trình tìm kiếm theo chiều sâu và quay lui, vét cạn, ứng dụng trong các bài toán tính toán biểu thức, ...

Một hình ảnh một stack

Ta có thể định nghĩa CTDL stack như sau: stack là một CTDL trừu tượng (ADT) tuyến tính hỗ trợ hai thao tác chính:

- Push(o): Thêm đối tượng o vào đầu stack
- Pop(): Lấy đối tượng ở đầu stack ra khỏi stack và trả về giá trị của nó. Nếu stack rỗng thì lỗi sẽ xảy ra.

Ngoài ra, stack cũng hỗ trợ một số thao tác khác:

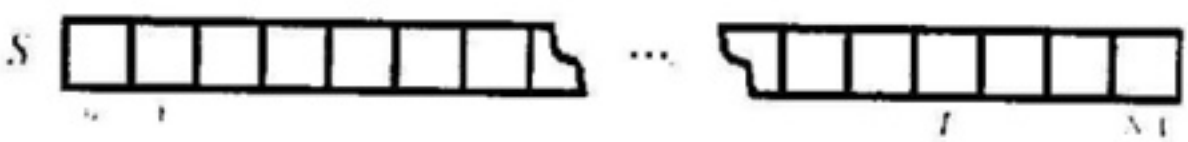
- isEmpty(): Kiểm tra xem stack có rỗng không.
- Top(): Trả về giá trị của phần tử nằm ở đầu stack mà không hủy nó khỏi stack. Nếu stack rỗng thì lỗi sẽ xảy ra.

Các thao tác thêm, trích và huỷ một phần tử chỉ được thực hiện ở cùng một phía của stack do đó hoạt động của stack được thực hiện theo nguyên tắc LIFO (Last In First Out - vào sau ra trước).

Để biểu diễn stack, ta có thể dùng mảng một chiều hoặc dùng danh sách liên kết.

Biểu diễn stack dùng mảng

- Ta có thể tạo một stack bằng cách khai báo một mảng một chiều với kích thước tối đa là N (ví dụ, N có thể bằng 1000).
- Như vậy stack có thể chứa tối đa N phần tử đánh số từ 0 đến N -1. Phần tử nằm ở đầu stack sẽ có chỉ số t (lúc đó trong stack đang chứa t+1 phần tử)
- Để khai báo một stack, ta cần một mảng một chiều S, biến nguyên t cho biết chỉ số của đầu stack và hằng số N cho biết kích thước tối đa của stack.



- Tạo stack S và quản lý đỉnh stack bằng biến t:

```
Data    S [N];
int     t;
```

Lệnh t = 0 sẽ tạo ra một stack S rỗng. Giá trị của Top sẽ cho biết số phần tử hiện hành có trong stack.

- Do khi cài đặt bằng mảng một chiều, stack có kích thước tối đa nên ta cần xây dựng thêm một thao tác phụ cho stack:

IsFull(): Kiểm tra xem stack có đầy chưa.

- Khi stack đầy, việc gọi đến hàm push() sẽ phát sinh ra lỗi.
- Sau đây là các thao tác tương ứng cho array-stack:
  - Kiểm tra stack rỗng hay không:

```
char IsEmpty()
{
```

```
    if(t == 0)        // stack rỗng
       return 1;

    else
       return 0;
}
```

- Kiểm tra stack rỗng hay không:

```
char IsFull()
{
    if(t >= N)        // stack đầy
       return 1;
    else
       return 0;
}
```

- Thêm một phần tử x vào stack S

```
void Push(Data x)
{
    if(t < N) // stack chưa đầy
    {
       S[t] = x; t++;
    }
    else  puts("Stack đầy")
}
```

- Trích thông tin và huỷ phần tử ở đỉnh stack S

```
Data Pop()
{  Data  x;

    if(t > 0)         // stack khác rỗng
    {
       t--;
       x = S[t];
       return   x;
    }
    else  puts("Stack rỗng")
```

```
}
```

- Xem thông tin của phần tử ở đỉnh stack S

```
Data Top()
{  Data  x;

   if(t > 0)          // stack khác rỗng
   {
      x = S[t-1];
      return   x;
   }
   else  puts("Stack rỗng")
}
```

- Các thao tác trên đều làm việc với chi phí O(1).
- Việc cài đặt stack thông qua mảng một chiều đơn giản và khá hiệu quả.
- Tuy nhiên, hạn chế lớn nhất của phương án cài đặt này là giới hạn về kích thước của stack N. Giá trị của N có thể quá nhỏ so với nhu cầu thực tế hoặc quá lớn sẽ làm lãng phí bộ nhớ.

Biểu diễn stack dùng danh sách

- Ta có thể tạo một stack bằng cách sử dụng một danh sách liên kết đơn (DSLK). Có thể nói, DSLK có những đặc tính rất phù hợp để dùng làm stack vì mọi thao tác trên stack đều diễn ra ở đầu stack.
- Sau đây là các thao tác tương ứng cho list-stack:
  - Tạo stack S rỗng

    ```
    LIST  * S;
    ```

    Lệnh `S.pHead=1.pTail= NULL` sẽ tạo ra một Stack S rỗng.

  - Kiểm tra stack rỗng :

```
char  IsEmpty(LIST &S)
{
   if (S.pHead == NULL) // stack rỗng
      return 1;
   else  return 0;
}
```

- Thêm một phần tử p vào stack S

```
void  Push(LIST &S, Data x)
{
   InsertHead(S, x);
}
```

- Trích huỷ phần tử ở đỉnh stack S

```
Data  Pop(LIST &S)
{  Data  x;

   if(isEmpty(S)) return NULLDATA;
   x = RemoveFirst(S);
   return x;
}
```

- Xem thông tin của phần tử ở đỉnh stack S

```
Data  Top(LIST &S)
{
   if(isEmpty(S)) return NULLDATA;
   return l.Head->Info;
}
```

Ứng dụng của stack

Cấu trúc stack thích hợp lưu trữ các loại dữ liệu mà trình tự truy xuất ngược với trình tự lưu trữ, do vậy một số ứng dụng sau thường cần đến stack :

- Trong trình biên dịch (thông dịch), khi thực hiện các thủ tục, stack được sử dụng để lưu môi trường của các thủ tục.

- Trong một số bài toán của lý thuyết đồ thị (như tìm đường đi), stack cũng thường được sử dụng để lưu dữ liệu khi giải các bài toán này.

- Ngoài ra, stack cũng còn được sử dụng trong trường hợp khử đệ qui đuôi. Ví dụ thủ tục Quick_Sort sau dùng stack để khử đệ qui:

1. l:=1; r:=n;
2. Chọn phần tử giữa x:=a[(l+r) div 2];
3. Phân hoạch (l,r) thành (l1,r1) và (l2,r2) bằng cách xét:

   - y thuộc (l1,r1) nếu y≤x;
   - y thuộc (l2,r2) ngược lại;

4. Nếu phân hoạch (l2,r2) có nhiều hơn một phần tử thực hiện:
   - Cất (l2,r2) vào stack;
   - Nếu (l1,r1) có nhiều hơn một phần tử thực hiện:
     - l=l1;
     - r=r1;
     - Goto 2;

   Ngược lại
     - Lấy (l,r) ra khỏi stack nếu stack khác rỗng;
     - và Goto 2;
     - Nếu không dừng;

Hàng đợi (Queue)

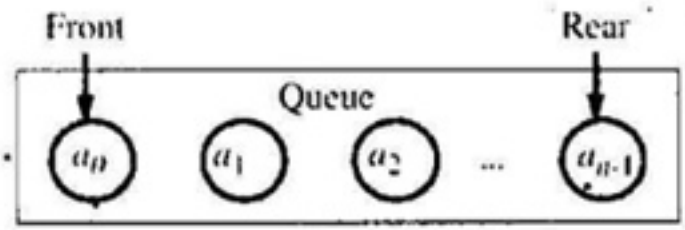
Hàng đợi là một vật chứa (container) các đối tượng làm việc theo cơ chế **FIFO** (**F**irst **I**n **F**irst **O**ut) nghĩa là việc thêm một đối tượng vào hàng đợi hoặc lấy một đối tượng ra khỏi hàng đợi được

thực hiện theo cơ chế "Vào trước ra trước".

Các đối tượng có thể được thêm vào hàng đợi bất kỳ lúc nào nhưng chỉ có đối tượng thêm vào đầu tiên mới được phép lấy ra khỏi hàng đợi.

Thao tác thêm một đối tượng vào hàng đợi và lấy một đối tượng ra khỏi hàng đợi lần lượt được gọi là "enqueue" và "dequeue".

Việc thêm một đối tượng vào hàng đợi luôn diễn ra ở cuối hàng đợi và một phần tử luôn được lấy ra từ đầu hàng đợi.



Trong tin học, CTDL hàng đợi có nhiều ứng dụng: khử đệ qui, tổ chức lưu vết các quá trình tìm kiếm theo chiều rộng và quay lui, vét cạn, tổ chức quản lý và phân phối tiến trình trong các hệ điều hành, tổ chức bộ đệm bàn phím, ...

Ta có thể định nghĩa CTDL hàng đợi như sau: hàng đợi là một CTDL trừu tượng (ADT) tuyến tính. Tương tự như stack, hàng đợi hỗ trợ các thao tác:

- EnQueue(o): Thêm đối tượng o vào cuối hàng đợi
- DeQueue(): Lấy đối tượng ở đầu queue ra khỏi hàng đợi và trả về giá trị của nó. Nếu hàng đợi rỗng thì lỗi sẽ xảy ra.
- IsEmpty(): Kiểm tra xem hàng đợi có rỗng không.
- Front(): Trả về giá trị của phần tử nằm ở đầu hàng đợi

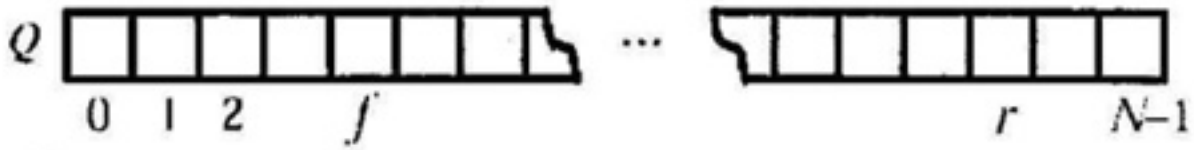
mà không hủy nó. Nếu hàng đợi rỗng thì lỗi sẽ xảy ra.

Các thao tác thêm, trích và huỷ một phần tử, phải được thực hiện ở hai phía khác nhau của hàng đợi, do đó hoạt động của hàng đợi được thực hiện theo nguyên tắc FIFO (First In First Out - vào trước ra trước).

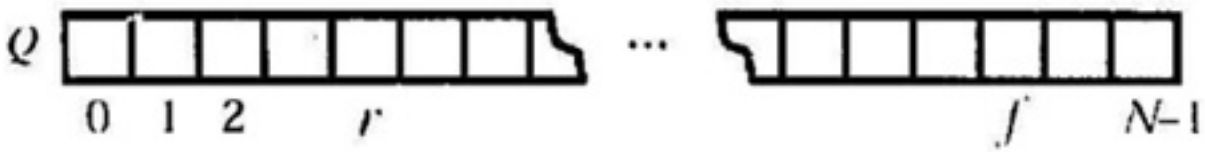
Cũng như stack, ta có thể dùng cấu trúc mảng một chiều hoặc cấu trúc danh sách liên kết để biểu diễn cấu trúc hàng đợi.

Biểu diễn dùng mảng

- Ta có thể tạo một hàng đợi bằng cách sử dụng một mảng một chiều với kích thước tối đa là N (ví dụ, N có thể bằng 1000) theo kiểu xoay vòng (coi phần tử a_{n-1} kề với phần tử a_0).
- Như vậy hàng đợi có thể chứa tối đa N phần tử. Phần tử nằm ở đầu hàng đợi (front element) sẽ có chỉ số f. Phần tử nằm ở cuối hàng đợi (rear element) sẽ có chỉ số r (xem hình).
- Để khai báo một hàng đợi, ta cần một mảng một chiều Q, hai biến nguyên f, r cho biết chỉ số của đầu và cuối của hàng đợi và hằng số N cho biết kích thước tối đa của hàng đợi. Ngoài ra, khi dùng mảng biểu diễn hàng đợi, ta cũng cần một giá trị đặc biệt để gán cho những ô còn trống trên hàng đợi. Giá trị này là một giá trị nằm ngoài miền xác định của dữ liệu lưu trong hàng đợi. Ta ký hiệu nó là NULLDATA như ở những phần trước.
- Trạng thái hàng đợi lúc bình thường:



- Trạng thái hàng đợi lúc xoay vòng:



Câu hỏi đặt ra: khi giá trị f = r cho ta điều gì ? Ta thấy rằng, lúc này hàng đợi chỉ có thể ở một trong hai trạng thái là rỗng hoặc đầy. Coi như một bài tập các bạn hãy tự suy nghĩ tìm câu trả lời trước khi đọc tiếp để kiểm tra kết quả.

- Hàng đợi có thể được khai báo cụ thể như sau:

```
Data  Q[N];
int   f, r;
```

- Cũng như strack, do khi cài đặt bằng mảng một chiều, hàng đợi có kích thước tối đa nên ta cần xây dựng thêm một thao tác phụ cho hàng đợi:

IsFull(): Kiểm tra xem hàng đợi có đầy chưa.

- Sau đây là các thao tác tương ứng cho array-queue:
  - Tạo hàng đợi rỗng:

```
void  InitQueue()
{
   f = r = 0;
   for(int i = 0; i < N; i++)
      Q[i] = NULLDATA;
}
```

  - Kiểm tra hàng đợi rỗng:

```
char  IsEmpty()
```

```
{
   return (Q[f] == NULLDATA);
}
```

- **Kiểm tra hàng đợi đầy:**

```
char , IsFull()
{
   return (Q[r] != NULLDATA);
}
```

- **Thêm phần tử x vào cuối hàng đợi Q**

```
char  EnQueue(Data x)
{
   if(IsFull()) return -1; //Queue đầy
   Q[r++] = x;
   if(r == N)   // xoay vòng
      r = 0;
}
```

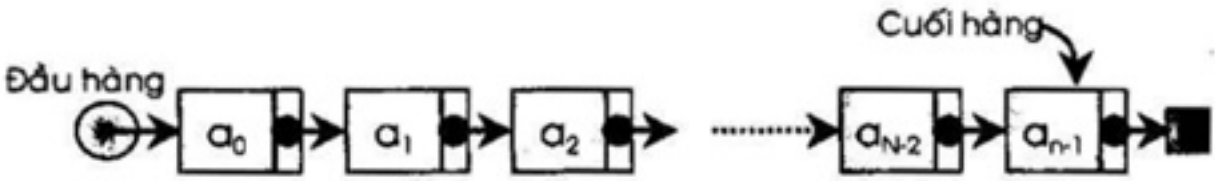
- **Trích, huỷ phần tử ở đầu hàng đợi Q**

```
Data  DeQueue()
{  Data x;

   if(IsEmpty()) return NULLDATA; //Queue rỗng
   x = Q[f]; Q[f++] = NULLDATA;
   if(f == N)   f = 0; // xoay vòng



   return x;
}
```

- **Xem thông tin của phần tử ở đầu hàng đợi Q**

```
Data  Front()
{
   if(IsEmpty()) return NULLDATA; //Queue rỗng
   return Q[f];
}
```

- Ta có thể tạo một hàng đợi bằng cách sử dụng một DSLK đơn.
- Phần tử đầu DSKL (head) sẽ là phần tử đầu hàng đợi, phần tử cuối DSKL (tail) sẽ là phần tử cuối hàng đợi.



- Sau đây là các thao tác tương ứng cho array-queue:
  - Tạo hàng đợi rỗng:

    Lệnh Q.pHead = Q.pTail = NULL sẽ tạo ra một hàng đợi rỗng.

  - Kiểm tra hàng đợi rỗng :

```
char  IsEmpty(LIST Q)
{
   if (Q.pHead == NULL) // stack rỗng
      return 1;
   else  return 0;
}
```

  - Thêm một phần tử p vào cuối hàng đợi

```
void  EnQueue(LIST Q, Data x)
{
   InsertTail(Q, x);
}
```

  - Trích/Hủy phần tử ở đầu hàng đợi

```
Data  DeQueue(LIST Q)
{  Data  x;

   if (IsEmpty(Q)) return NULLDATA;
   x = RemoveFirst(Q);
```

```
    return x;
}
```

- Xem thông tin của phần tử ở đầu hàng đợi

```
Data  Front(LIST Q)
{
    if (IsEmpty(Q)) return NULLDATA;
    return Q.pHead->Info;
}
```

- Các thao tác trên đều làm việc với chi phí O(1).
- Lưu ý, nếu không quản lý phần tử cuối xâu, thao tác dequeue sẽ có độ phức tạp O(n).

Ứng dụng của hàng đợi

Hàng đợi có thể được sử dụng trong một số bài toán:

- Bài toán 'sản xuất và tiêu thụ' (ứng dụng trong các hệ điều hành song song).
- Bộ đệm (ví dụ: Nhấn phím -> Bộ đệm -> CPU xử lý).
- Xử lý các lệnh trong máy tính (ứng dụng trong hệ điều hành, trình biên dịch), hàng đợi các tiến trình chờ được xử lý, ...

V. MỘT SỐ CẤU TRÚC DỮ LIỆU DẠNG DANH SÁCH LIÊN KẾT KHÁC

1. Danh sách liên kết kép

Danh sách liên kết kép là danh sách mà mỗi phần tử trong danh sách có kết nối với một phần tử đứng trước và một phần tử đứng sau nó.

Các khai báo sau định nghiã một danh sách liên kết kép đơn giản trong đó ta dùng hai con trỏ: pPrev liên kết với phần tử đứng trước và pNext như thường lệ, liên kết với phần tử đứng sau:

```
typedef  struct  tagDNode
{
   Data  Info;
   struct tagDNode* pPre;   // trỏ đến phần tử đứng trước
   struct tagDNode* pNext;  // trỏ đến phần tử đứng sau
}DNODE;

typedef  struct  tagDList
{
   DNODE* pHead;   // trỏ đến phần tử đứng trước
   DNODE* pTail;   // trỏ đến phần tử đứng sau
}DLIST;
```

khi đó, thủ tục khởi tạo một phần tử cho danh sách liên kết kép được viết lại như sau :

```
DNODE*   GetNode(Data x)
{  DNODE *p;

   // Cấp phát vùng nhớ cho phần tử
```

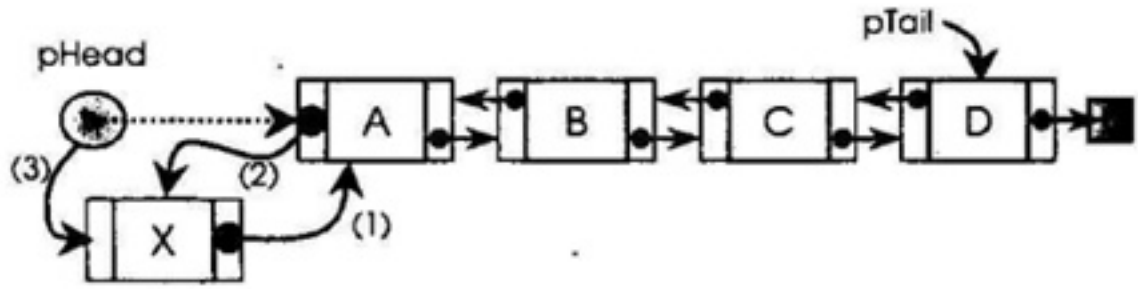
```
    p = new DNODE;
    if ( p==NULL)    {
       printf("Không đủ bộ nhớ");
       exit(1);
    }
    // Gán thông tin cho phần tử p
    p ->Info = x;
    p->pPrev = NULL;
    p->pNext = NULL;
    return p;
}
```

Tương tự danh sách liên kết đơn, ta có thể xây dựng các thao tác cơ bản trên danh sách liên kết kép (xâu kép). Một số thao tác không khác gì trên xâu đơn. Dưới đây là một số thao tác đặc trưng của xâu kép:

Chèn một phần tử vào danh sách

Có bốn loại thao tác chèn new_ele vào danh sách:

Cách 1: Chèn vào đầu danh sách

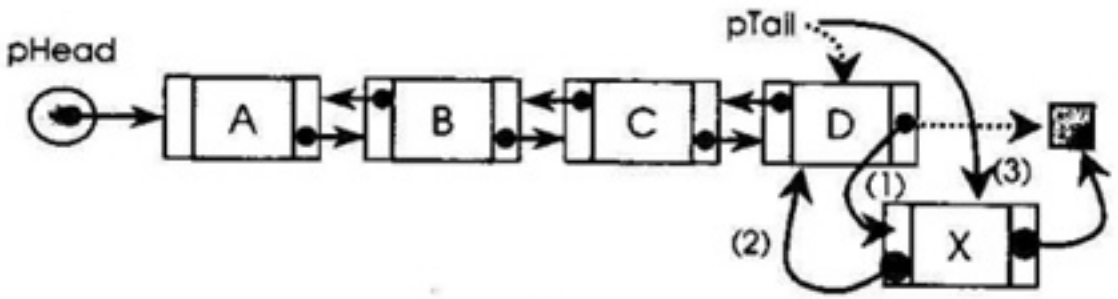


Cài đặt

```
void AddFirst(DLIST &l, DNODE* new_ele)
{
    if (l.pHead==NULL)  //Xâu rỗng
    {
       l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {
       new_ele->pNext = l.pHead;                    //(1)
```

```
        l.pHead ->pPrev = new_ele;          // (2)
        l.pHead = new_ele;                  // (3)
    }
}
NODE* InsertHead(DLIST &l, Data x)
{   NODE* new_ele = GetNode(x);

    if (new_ele ==NULL)  return NULL;
    if (l.pHead==NULL)
    {
        l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {
        new_ele->pNext = l.pHead;           // (1)
        l.pHead ->pPrev = new_ele;          // (2)
        l.pHead = new_ele;                  // (3)
    }
    return new_ele;
}
```

Cách 2 : Chèn vào cuối danh sách

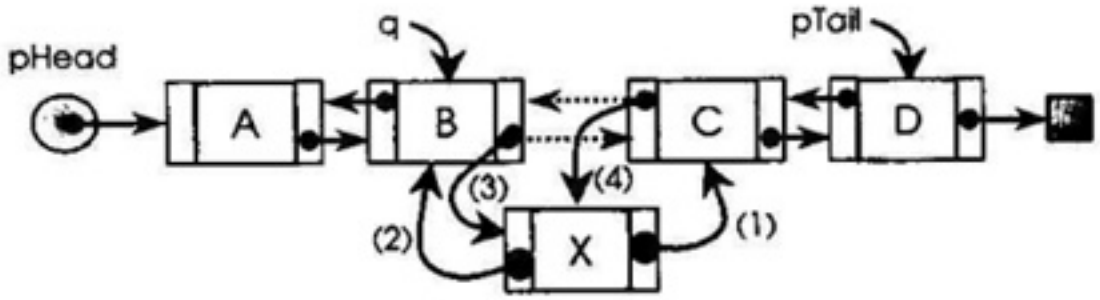


Cài đặt

```
void AddTail(DLIST &l, DNODE *new_ele)
{
    if (l.pHead==NULL)
    {
        l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {
        l.pTail->Next = new_ele;            // (1)
```

```
        new_ele ->pPrev = l.pTail;          // (2)
        l.pTail = new_ele;                  // (3)
    }
}

NODE* InsertTail(DLIST &l, Data x)
{  NODE* new_ele = GetNode(x);

    if (new_ele ==NULL)  return NULL;
    if (l.pHead==NULL)
    {
        l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {
        l.pTail->Next = new_ele;            // (1)
        new_ele ->pPrev = l.pTail;          // (2)
        l.pTail = new_ele;                  // (3)
    }
    return new_ele;
}
```

Cách 3 : Chèn vào danh sách sau một phần tử q

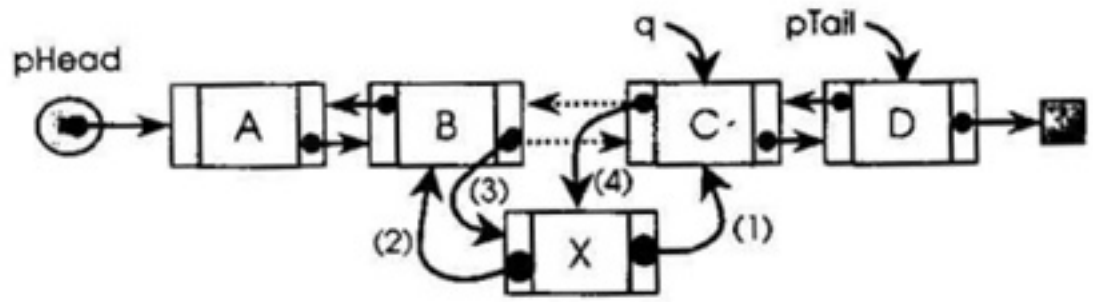


Cài đặt

```
void AddAfter(DLIST &l, DNODE* q,DNODE*
new_ele)
{  DNODE* p = q->pNext;

    if ( q!=NULL)
    {
        new_ele->pNext = p;                 //(1)
        new_ele->pPrev = q;                 //(2)
```

```
        q->pNext = new_ele;                          //(3)
        if(p != NULL)
            p->pPrev = new_ele;                      //(4)
        if(q == l.pTail)
            l.pTail = new_ele;
    }
    else  //chèn vào đầu danh sách
        AddFirst(l, new_ele);
}

void InsertAfter(DLIST &l, DNODE *q, Data x)
{
    DNODE* p = q->pNext;
    NODE* new_ele = GetNode(x);

    if (new_ele ==NULL)  return NULL;

    if ( q!=NULL)
    {
        new_ele->pNext = p;                          //(1)
        new_ele->pPrev = q;                          //(2)
        q->pNext = new_ele;                          //(3)
        if(p != NULL)
            p->pPrev = new_ele;                      //(4)
        if(q == l.pTail)
            l.pTail = new_ele;
    }
    else  //chèn vào đầu danh sách
        AddFirst(l, new_ele);
}
```

Cách 4 : Chèn vào danh sách trước một phần tử q



Cài đặt

```
void AddBefore(DLIST &l, DNODE q, DNODE*
new_ele)
{  DNODE* p = q->pPrev;
   if ( q!=NULL)
   {
      new_ele->pNext = q;                    //(1)
      new_ele->pPrev = p;                    //(2)
      q->pPrev = new_ele;                    //(3)
      if(p != NULL)
         p->pNext = new_ele;                 //(4)
      if(q == l.pHead)
         l.pHead = new_ele;
   }
   else  //chèn vào đầu danh sách
      AddTail(l, new_ele);
}

void InsertBefore(DLIST &l, DNODE q, Data x)
{  DNODE* p = q->pPrev;
   NODE* new_ele = GetNode(x);

   if (new_ele ==NULL)  return NULL;

   if ( q!=NULL)
   {
      new_ele->pNext = q;                    //(1)
      new_ele->pPrev = p;                    //(2)
      q->pPrev = new_ele;                    //(3)
      if(p != NULL)
         p->pNext = new_ele;                 //(4)
      if(q == l.pHead)
         l.pHead = new_ele;
   }
   else  //chèn vào đầu danh sách
      AddTail(l, new_ele);
}
```

Hủy một phần tử khỏi danh sách

Có năm loại thao tác thông dụng hủy một phần tử ra khỏi xâu. Chúng ta sẽ lần lượt khảo sát chúng.

Hủy phần tử đầu xâu

```
Data RemoveHead(DLIST &l)
{  DNODE *p;
   Data  x = NULLDATA;

   if ( l.pHead != NULL)
   {
      p = l.pHead; x = p->Info;
      l.pHead = l.pHead->pNext;
      l.pHead->pPrev = NULL;
      delete p;
      if(l.pHead == NULL)  l.pTail = NULL;
      else l.pHead->pPrev = NULL;
   }
   return x;
}
```

Hủy phần tử cuối xâu

```
Data RemoveTail(DLIST &l)
{  DNODE *p;
   Data  x = NULLDATA;

   if ( l.pTail != NULL)
   {
      p = l.pTail; x = p->Info;
      l.pTail = l.pTail->pPrev;
      l.pTail->pNext = NULL;
      delete p;
      if(l.pHead == NULL)  l.pTail = NULL;
      else l.pHead->pPrev = NULL;
   }
   return x;
}
```

Hủy một phần tử đứng sau phần tử q

```
void RemoveAfter (DLIST &l, DNODE *q)
{  DNODE *p;

   if ( q != NULL)
   {
      p = q ->pNext ;
      if ( p != NULL)
      {
         q->pNext = p->pNext;
         if(p == l.pTail)  l.pTail = q;
         else p->pNext->pPrev = q;
         delete p;
      }
   }
   else
      RemoveHead(l);
}
```

Hủy một phần tử đứng trước phần tử q

```
void RemoveAfter (DLIST &l, DNODE *q)
{  DNODE *p;

   if ( q != NULL)
   {
      p = q ->pPrev;
      if ( p != NULL)
      {
         q->pPrev = p->pPrev;
         if(p == l.pHead)  l.pHead = q;
         else p->pPrev->pNext = q;
         delete p;
      }
   }
   else
      RemoveTail(l);
}
```

Hủy một phần tử có khoá k

```
int RemoveNode(DLIST &l, Data k)
{
   DNODE *p = l.pHead;
   NODE  *q;

   while( p != NULL)
   {
      if(p->Info == k) break;
      p = p->pNext;
   }
   if(p == NULL) return 0; //Không tìm thấy k
   q = p->pPrev;
   if ( q != NULL)
   {
      p = q ->pNext ;
      if ( p != NULL)
      {
         q->pNext = p->pNext;
         if(p == l.pTail)
            l.pTail = q;
         else p->pNext->pPrev = q;
      }
   }
   else //p là phần tử đầu xâu
   {
      l.pHead = p->pNext;
      if(l.pHead == NULL)
      l.pTail = NULL;
      else
         l.pHead->pPrev = NULL;
   }
   delete p;
   return 1;
}
```

Sắp xếp danh sách

Việc sắp xếp danh sách trên xâu kép về cơ bản không khác gì trên xâu đơn. Ta chỉ cần lưu ý một điều duy nhất là cần bảo toàn các mối liên kết hai chiều trong khi sắp xếp.

Ở đây, như một ví dụ, chúng ta sẽ xét cài đặt của thuật toán Quick sort trên xâu kép:

```
void  DListQSort(DLIST & l)
{  DNODE *p, *X;   // X chỉ đến phần tử cầm canh
   DLIST l1, l2;

   if(l.pHead == l.pTail) return;//đã có thứ tự
   l1.pHead == l1.pTail = NULL; //khởi tạo
   l2.pHead == l2.pTail = NULL;
   X = l.pHead; l.pHead = X->pNext;
   while(l.pHead != NULL) //Tách l thành l1, l2;
   {
      p = l.pHead;
      l.pHead = p->pNext; p->pNext = NULL;
      if (p->Info <= X->Info)
         AddTail(l1, p);
      else
         AddTail(l2, p);
   }
   DListQSort(l1);//Gọi đệ qui để sort l1
   DListQSort(l2);//Gọi đệ qui để sort l2
   //Nối l1, X và l2 lại thành l đã sắp xếp.
   if(l1.pHead != NULL)
   {
      l.pHead = l1.pHead; l1.pTail->pNext = X;
      X->pPrev = l1.pTail;
   }
   else
      l.pHead = X;
   X->pNext = l2;
   if(l2.pHead != NULL)
   {
      l.pTail = l2.pTail;
      l2->pHead->pPrev = X;
   }
   else
      l.pTail = X;
}
```

Xâu kép về mặt cơ bản có tính chất giống như xâu đơn. Tuy nhiên nó có một số tính chất khác xâu đơn như sau:

- Xâu kép có mối liên kết hai chiều nên từ một phần tử bất kỳ có thể truy xuất một phần tử bất kỳ khá; trong khi trên xâu đơn ta chỉ có thể truy xuất đến các phần tử đứng sau một phần tử cho trước. Điều này dẫn đến việc ta có thể dễ dàng hủy phần tử cuối xâu kép, còn trên xâu đơn thao tác này tốn chi phí O(n).

- Bù lại, xâu kép tốn chi phí gấp đôi so với xâu đơn cho việc lưu trữ các mối liên kết. Điều này khiến việc cập nhật cũng nặng nề hơn trong một số trường hợp. Như vậy ta cần cân nhắc lựa chọn CTDL hợp lý khi cài đặt cho một ứng dụng cụ thể.

2. Hàng đợi hai đầu (double-ended queue)

Hàng đợi hai đầu (gọi tắt là Deque) là một vật chứa các đối tượng mà việc thêm hoặc hủy một đối tượng được thực hiện ở cả hai đầu của nó.

Ta có thể định nghĩa CTDL deque như sau: deque là một CTDL trừu tượng (ADT) hỗ trợ các thao tác chính sau:

- InsertFirst(e): Thêm đối tượng e vào đầu deque
- InsertLast(e): Thêm đối tượng e vào cuối deque
- RemoveFirst(): Lấy đối tượng ở đầu deque ra khỏi deque và trả về giá trị của nó.
- RemoveLast(): Lấy đối tượng ở cuối deque ra khỏi deque và trả về giá trị của nó.

Ngoài ra, deque cũng hỗ trợ các thao tác sau:

- IsEmpty(): Kiểm tra xem deque có rỗng không.

- First(): Trả về giá trị của phần tử nằm ở đầu deque mà không hủy nó.
- Last(): Trả về giá trị của phần tử nằm ở cuối deque mà không hủy nó.

Dùng deque để cài đặt stack và queue

Ta có thể dùng deque để biểu diễn stack. Khi đó ta có các thao tác tương ứng như sau:

STT	Stack	Deque
1	Push	InsertLast
2	Pop	RemoveLast
3	Top	Last
4	IsEmpty	IsEmpty

Tương tự, ta có thể dùng deque để biểu diễn queue. Khi đó ta có các thao tác tương ứng như sau:

STT	Queue	Deque
1	Enqueue	InsertLast
2	Dequeue	RemoveFist
3	Front	First
4	IsEmpty	IsEmpty

Cài đặt deque

Do đặc tính truy xuất hai đầu của deque, việc xây dựng CTDL biểu diễn nó phải phù hợp.

Ta có thể cài đặt CTDL deque bằng danh sách liên kết đơn.

Tuy nhiên, khi đó thao tác RemoveLast hủy phần tử ở cuối deque sẽ tốn chi phí O(n). Điều này làm giảm hiệu quả của CTDL. Thích hợp nhất để cài đặt deque là dùng danh sách liên kết kép. Tất cả các thao tác trên deque khi đó sẽ chỉ tốn chi phí O(1).

3. Danh sách liên kết có thứ tự (Odered List)

Danh sách liên kết có thứ tự (gọi tắt là OList) là một vật chứa các đối tượng theo một trình tự nhất định. Trình tự này thường là một khóa sắp xếp nào đó. Việc thêm một đối tượng vào OList phải bảo đảm tôn trọng thứ tự này.

Ta có thể cài đặt OList bằng DSLK đơn hoặc DSLK đôi với việc định nghĩa lại duy nhất một phép thêm phần tử: thêm bảo toàn thứ tự; nghĩa là trên OList chỉ cho phép một thao tác thêm phần tử sao cho thứ tự định nghĩa trên OList phải bảo toàn.

Ví dụ, khi cài đặt OList bằng xâu đơn, hàm thêm một phần tử có thể được xây dựng như sau:

```
NODE* InsertNode(LIST & l, Data X)
{  Node* q  = NULL, *p = l.pHead;

   if(IsEmpty(l)) return InsertHead(l, X);
while(p) {
      if(p->Info >= X)  break;
      q  = p; p   = q->pNext;
   }

   Node*pT = new(Node);

   pT->Info = X; pT->pNext= p;
   if(q)     q->pNext = pT;
   else      l.pHead  = pT;
   if(q == l->pTail) l.pTail  = pT;
   return pT;
```

```
}
```

Khi đó, hàm tìm kiếm một phần tử được viết lại như sau:

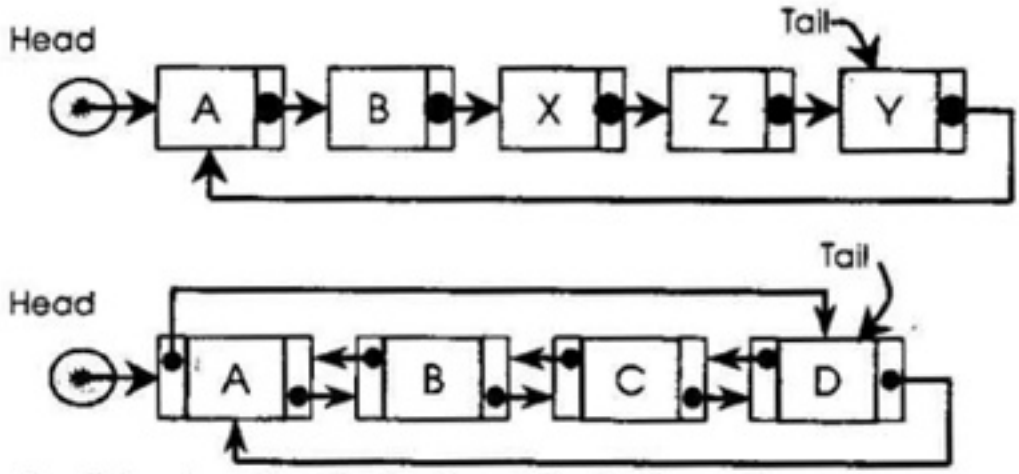
```
}
NODE* SearchNode(LIST l, Data X)
{  Node*p = l.pHead;

while(p) {
      if(p->Info == X)  break;
      else if(p->Info > X) return NULL;
      p  = p->pNext;
   }
   return p;
}
```

Ta có thể dùng OList để cài đặt CTDL hàng đợi có độ ưu tiên. Trong hàng đợi có độ ưu tiên, mỗi phần tử được gán cho một độ ưu tiên. Hàng đợi có độ ưu tiên cũng giống như hàng đợi bình thường ở thao tác lấy một phần tử khỏi hàng đợi (lấy ở đầu queue) nhưng khác ở thao tác thêm vào. Thay vì thêm vào ở cuối queue, việc thêm vào trong hàng đợi có độ ưu tiên phải bảo đảm phần tử có độ ưu tiên cao đứng trước, phần tử có độ ưu tiên thấp đứng sau. Hàng đợi có độ ưu tiên có nhiều ứng dụng. Ví dụ, CTDL này có thể dùng để quản lý hàng đợi các tiến trình chờ được xử lý trong các hệ điều hành đa nhiệm.

4. Danh sách liên kết vòng

Danh sách liên kết vòng (xâu vòng) là một danh sách đơn (hoặc kép) mà phần tử cuối danh sách thay vì mang giá trị NULL, trỏ tới phần tử đầu danh sách. Để biểu diễn, ta có thể xử dụng các kỹ thuật biểu diễn như danh sách đơn (hoặc kép).

Ta có thể khai báo xâu vòng như khai báo xâu đơn (hoặc kép).



Trên danh sách vòng ta có các thao tác thường gặp sau:

Tìm phần tử trên danh sách vòng

Danh sách vòng không có phần tử đầu danh sách rõ rệt, nhưng ta có thể đánh dấu một phần tử bất kỳ trên danh sách xem như phần tử đầu xâu để kiểm tra việc duyệt đã qua hết các phần tử của danh sách hay chưa.

```
NODE* Search(LIST &l, Data  x)
{
   NODE  *p;

   p = l.pHead;
   do
   {
      if ( p->Info == x)
         return p;
      p = p->pNext;
   }while  (p != l.pHead); // chưa đi giáp vòng
   return p;
}
```

Thêm phần tử đầu xâu

```
void  AddHead(LIST &l, NODE *new_ele)
```

```
{
   if(l.pHead == NULL) //Xâu rỗng
   {
      l.pHead = l.pTail = new_ele;
      l.pTail->pNext = l.pHead;
   }
   else
   {
      new_ele->pNext = l.pHead;
      l.pTail->pNext = new_ele;
      l.pHead = new_ele;
   }
}
```

Thêm phần tử cuối xâu

```
void  AddTail(LIST &l, NODE *new_ele)
{
   if(l.pHead == NULL) //Xâu rỗng
   {
      l.pHead = l.pTail = new_ele;
      l.pTail->pNext = l.pHead;
   }
   else
   {
      new_ele->pNext = l.pHead;
      l.pTail->pNext = new_ele;
      l.pTail = new_ele;
   }
}
```

Thêm phần tử sau nút q

```
void  AddAfter(LIST &l, NODE *q, NODE *new_ele)
{
   if(l.pHead == NULL) //Xâu rỗng
   {
      l.pHead = l.pTail = new_ele;
      l.pTail->pNext = l.pHead;
   }
   else
   {
      new_ele->pNext = q->pNext;
```

```
        q->pNext = new_ele;
        if(q == l.pTail)
            l.pTail = new_ele;
    }
}
```

Hủy phần tử đầu xâu

```
void RemoveHead(LIST &l)
{  NODE  *p = l.pHead;

   if(p == NULL) return;
   if (l.pHead = l.pTail)   l.pHead = l.pTail = NULL;
   else
   {
      l.pHead = p->Next;
      if(p == l.pTail)
         l.pTail->pNext = l.pHead;
   }
   delete p;
}
```

Hủy phần tử đứng sau nút q

```
void RemoveAfter(LIST &l, NODE *q)
{  NODE  *p;

   if(q != NULL)
   {
      p = q ->Next ;
      if ( p == q)   l.pHead = l.pTail = NULL;
      else
      {
         q->Next = p->Next;
         if(p == l.pTail)
            l.pTail = q;
      }
      delete p;
   }
}
```

! LƯU Ý

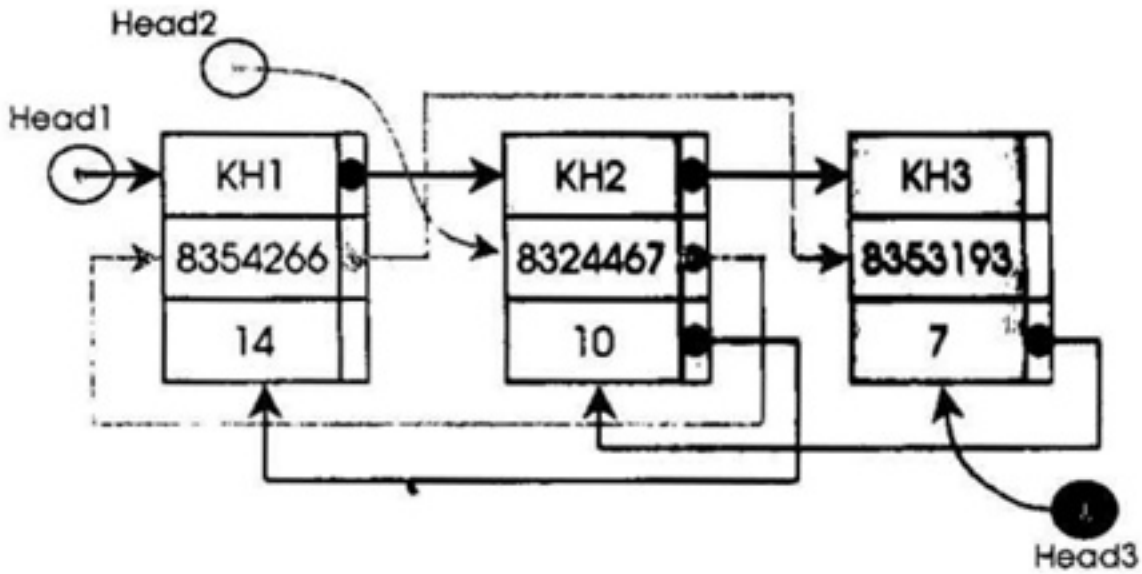
Đối với danh sách vòng, có thể xuất phát từ một phần tử bất kỳ để duyệt toàn bộ danh sách.

5. Danh sách có nhiều mối liên kết

Danh sách có nhiều mối liên kết là danh sách mà mỗi phần tử có nhiều khoá và chúng được liên kết với nhau theo từng loại khoá.

Danh sách có nhiều mối liên kết thường được xử dụng trong các ứng dụng quản lý một cơ sở dữ liệu lớn với những nhu cầu tìm kiếm dữ liệu theo những khoá khác nhau.

Ví dụ: Để quản lý danh mục điện thoại thuận tiện cho việc in danh mục theo những trình tự khác nhau: tên khách tăng dần, theo số điện thoại tăng dần, thời gian lắp đặt giảm dần, ta có thể tổ chức dữ liệu như hình trên: một danh sách với ba mối liên kết: một cho họ tên khách hàng, một cho số điện thoại và một cho thời gian lắp đặt.



Các thao tác trên một danh sách nhiều mối liên kết được tiến hành tương tự như trên danh sách đơn nhưng được thực hiện làm

nhiều lần và mỗi lần cho một liên kết.

6. Danh sách tổng quát

Danh sách tổng quát là một danh sách mà mỗi phần tử của nó có thể lại là một danh sách khác. Các ví dụ sau minh hoạ các cấu trúc danh sách tổng quát. Các thao tác trên một danh sách được xây dựng dựa trên cơ sở các thao tác trên danh sách liên kết chúng ta đã khảo sát.

Ví dụ 1

```
typedef  struct tagNode {
   DATA            Info;
   struct tagNode *pNext;
   void            *Sublist;
}NODE;
typedef  NODE  *PNODE;
```

Ví dụ 2

```
typedef  struct   tagNguoi {
   char  Ten[35];
   char  GioiTinh;
   intNamSinh;
   struct tagNguoi*Cha, *Me;
   struct tagNguoi*Anh, *Chi, *Em;
}NGUOI;
typedef  NGUOI *PNGUOI;
```

Ví dụ 3: Đồ thị (Graph)

Đồ thị là một CTDL liên kết tổng quát. Cấu trúc này bao gồm một tập hợp các nút. Những nút này liên kết với nhau một cách tùy

ý. Một nút có thể nối đến một số lượng tùy ý các nút khác và ngược lại:

```
typedef  struct tagGNode   {
   DATA            Info;
   struct tagGNode**pLink;
}GNODE;
typedef GNODE  *PGNODE;
```

TÓM TẮT

Trong chương này, chúng ta đã xem xét các khái niệm về cấu trúc dữ liệu động, kiểu dữ liệu con trỏ, ... Các kiểu dữ liệu này có đặc tính mềm dẻo hơn hẳn các CTDL tĩnh. Chúng cho phép ta sử dụng bộ nhớ hiệu quả hơn.

Dạng đơn giản nhất của các CTDL động là danh sách liên kết. Đây là các CTDL tuyến tính. Ở phần đầu chương, chúng ta xem xét các danh sách liên kết đơn. Điểm đặc trưng của CTDL này là khả năng truy xuất tuần tự. Điều này làm cho việc truy vấn thông tin lưu trên CTDL loại này thường chậm. Tuy nhiên, cũng như các CTDL động khác, chúng cho phép sử dụng bộ nhớ hiệu quả hơn. Điều này thể hiện rõ qua các thao tác trên danh sách liên kết đơn mà chúng ta đã khảo sát.

Trên CTDL danh sách liên kết đơn, chúng ta đã thấy các thuật toán Merge sort, Quick sort và Radix sort thể hiện rất tốt bản chất của mình. Chúng là các thuật toán rất hiệu quả trên danh sách liên kết. Điều này làm rõ hơn mối quan hệ mật thiết giữa CTDL và các thuật toán trên đó.

Stack là một trong những dạng cài đặt ứng dụng cụ thể của danh sách liên kết. Stack là một đối tượng rất thông dụng trong thực tế và là một CTDL thông dụng trong lập trình. Khả năng ứng dụng của nó đã được nêu rõ.

Hàng đợi là một thể hiện khác của danh sách liên kết. Vai trò của nó trong lập trình cũng tương tự như stack. Chúng chỉ làm việc theo các cơ chế khác nhau. Nếu như stack làm việc theo cơ chế LIFO thì hàng đợi làm việc theo cơ chế FIFO.

Các dạng danh sách liên kết khác cũng đã được khảo sát. Danh sách kép và danh sách vòng là hai dạng danh sách liên kết thông dụng khác. Chúng được dùng thay cho danh sách liên kết đơn khi có nhu cầu.

Phần cuối của chương có giới thiệu qua về dạng tổng quát nhất của danh sách liên kết. Dạng này phức tạp nên thường không thông dụng.

Bài tập lý thuyết

27. Phân tích ưu, khuyết điểm của xâu liên kết so với mảng. Tổng quát hóa các trường hợp nên dùng xâu liên kết.

28. Xây dựng một cấu trúc dữ liệu thích hợp để biểu diễn đa thức P(x) có dạng :

$$P(x) = c_1x^{n1} + c_2x^{n2} + ... + c_kx^{nk}$$

Biết rằng:

- Các thao tác xử lý trên đa thức bao gồm :
  - thêm một phần tử vào cuối đa thức
  - in danh sách các phần tử trong đa thức theo :
    - thứ tự nhập vào
    - ngược với thứ tự nhập vào
  - hủy một phần tử bất kỳ trong danh sách
- Số lượng các phần tử không hạn chế
- Chỉ có nhu cầu xử lý đa thức trong bộ nhớ chính.

Giải thích lý do chọn CTDL đã định nghĩa.

Viết chương trình con ước lượng giá trị của đa thức P(x) khi biết x.

Viết chương trình con rút gọn biểu thức (gộp các phần tử cùng số mũ).

29. Xét đoạn chương trình tạo một xâu đơn gồm bốn phần tử (không quan tâm dữ liệu) sau đây:

```
Dx = NULL; p=Dx;
Dx = new (NODE);
for(i=0; i < 4; i++)
{
   p = p->next;
   p = new (NODE);
}
p->next = NULL;
```

Đoạn chương trình có thực hiện được thao tác tạo nêu trên không ? Tại sao ? Nếu không thì có thể sửa lại như thế nào cho đúng ?

30. Một ma trận chỉ chứa rất ít phần tử với giá trị có nghĩa (ví dụ: phần tử $\neq 0$) được gọi là ma trận thưa.

Ví dụ :

$$\begin{pmatrix} 0 & 0 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \end{pmatrix}$$

Dùng cấu trúc xâu liên kết để tổ chức biểu diễn một ma trận thưa sao cho tiết kiệm nhất (chỉ lưu trữ các phần tử có nghĩa).

Viết chương trình cho phép nhập, xuất ma trận.

Viết chương trình con cho phép cộng hai ma trận.

31. Bài toán Josephus : có N người đã quyết định tự sát tập thể bằng cách đứng trong vòng tròn và giết người thứ M quanh

vòng tròn, thu hẹp hàng ngũ lại khi từng người lần lượt ngã khỏi vòng tròn. Vấn đề là tìm ra thứ tự từng người bị giết.

Ví dụ : N = 9, M = 5 thì thứ tự là 5, 1, 7, 4, 3, 6, 9, 2, 8

Hãy viết chương trình giải quyết bài toán Josephus, xử dụng cấu trúc xâu liên kết.

32. Hãy cho biết nội dung của stack sau mỗi thao tác trong dãy :

EAS*Y**QUE***ST***I*ON

Với một chữ cái tượng trưng cho thao tác thêm chữ cái tương ứng vào stack, dấu * tượng trưng cho thao tác lấy nội dung một phần tử trong stack in lên màn hình.

Hày cho biết sau khi hoàn tất chuỗi thao tác, những gì xuất hiện trên màn hình ?

33. Hãy cho biết nội dung của hàng đợi sau mỗi thao tác trong dãy :

EAS*Y**QUE***ST***I*ON

Với một chữ cái tượng trưng cho thao tác thêm chữ cái tương ứng vào hàng đợi, dấu * tượng trưng cho thao tác lấy nội dung một phần tử trong hàng đợi in lên màn hình.

Hày cho biết sau khi hoàn tất chuỗi thao tác, những gì xuất hiện trên màn hình ?

34. Giả sử phải xây dựng một chương trình soạn thảo văn bản, hãy chọn cấu trúc dữ liệu thích hợp để lưu trữ văn bản trong quá trình soạn thảo. Biết rằng :

- Số dòng văn bản không hạn chế.
- Mỗi dòng văn bản có chiều dài tối đa 80 ký tự.
- Các thao tác yêu cầu gồm :
  + Di chuyển trong văn bản (lên, xuống, qua trái, qua phải)
  + Thêm, xoá sửa ký tự trong một dòng
  + Thêm, xoá một dòng trong văn bản
  + Đánh dấu, sao chép khối

Giải thích lý do chọn cấu trúc dữ liệu đó.

35. Viết hàm ghép hai xâu vòng L_1, L_2 thành một xâu vòng L với phần tử đầu xâu là phần tử đầu xâu của L_1.

Bài tập thực hành

36. Cài đặt thuật toán sắp xếp Chèn trực tiếp trên xâu kép. Có phát huy ưu thế của thuật toán hơn trên mảng hay không ?

37. Cài đặt thuật toán Quick sort theo kiểu không đệ qui.

38. Cài đặt thuật toán Merge sort trên xâu kép.

39. Cài đặt lại chương trình quản lý nhân viên theo bài tập 6 chương 1, nhưng sử dụng cấu trúc dữ liệu xâu liên kết. Biết rằng số nhân viên không hạn chế.

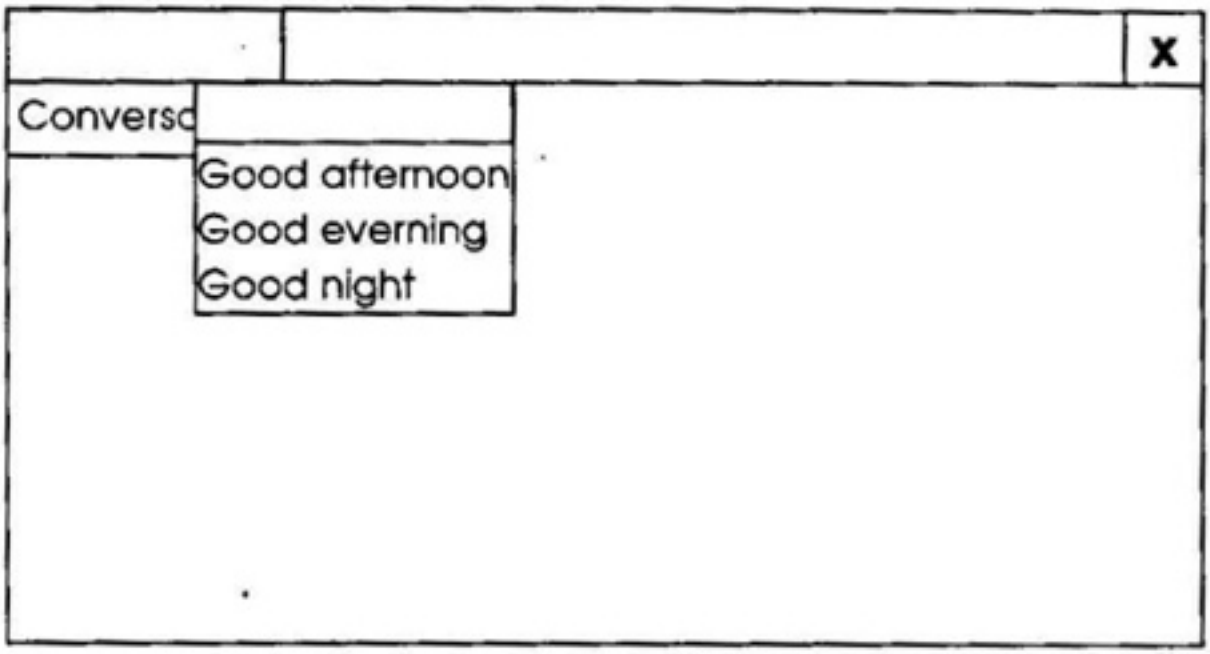
40. Cài đặt một chương trình soạn thảo văn bản theo mô tả trong bài tập 8.

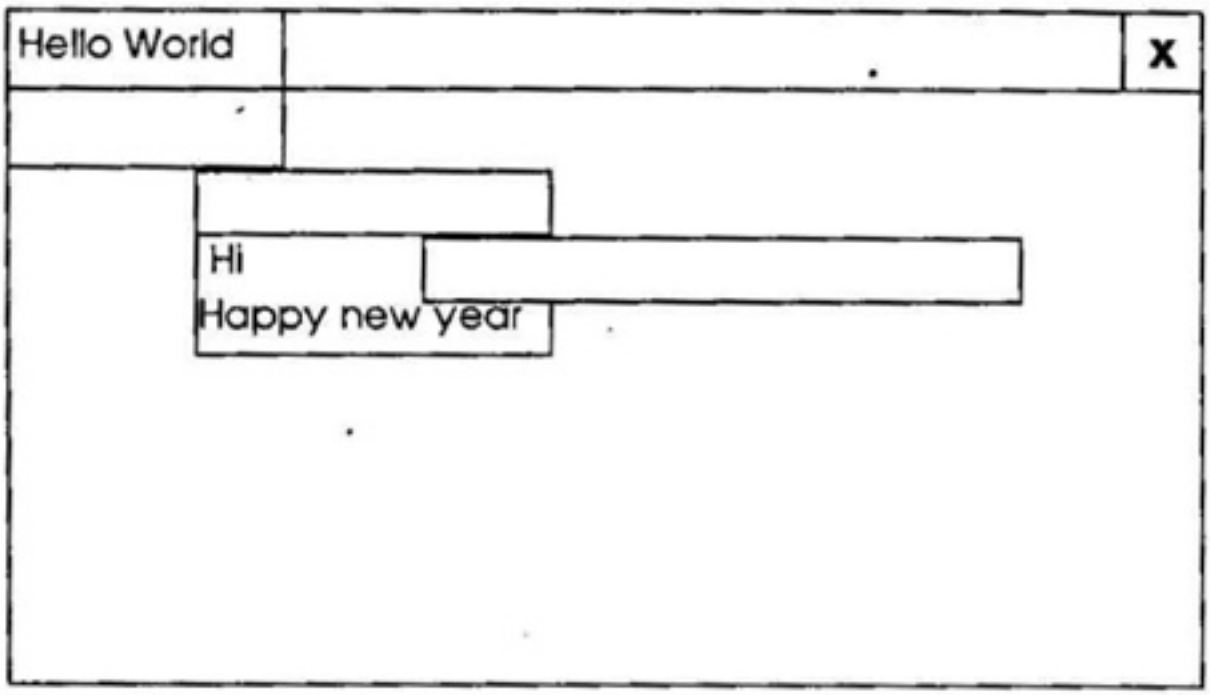
41. Cài đặt chương trình phát sinh hệ thống thực đơn cho một ứng dụng bất kỳ tùy theo mô tả của ứng dụng.

Ví dụ : Cho tập tin MENU.TXT chứa văn bảng có dạng sau :

```
Menu
popup
    item "Hello World" popup
        item "Good morning"
        item "Good afternoon"
        item "Good everning"
        item "Good night"
    end
    item "Conversation" popup          // menu cấp 1
        item "Good Luck" popup         // menu cấp 2
            item "Good luck for this examination"
        end
        item "Hi"
        item "Happy New Year"
    end
end
```

Chương trình sẽ đọc nội dung tập tin MENU.TXT và phát sinh giao diện sau :





42. Cài đặt chương trình tạo một bảng tính cho phép thực hiện các phép tính +, -, *, /, div trên các số có tối đa 30 chữ số, có chức năng nhớ (M+, M-, MC, MR).

43. *Cài đặt chương trình cho phép nhận vào một biểu thức gồm các số, các toán tử +, -, *, /, %, các hàm toán học sin, cos, tan, ln, e^x, dấu mở, đóng ngoặc "(", ")" và tính toán giá trị của biểu thức này.

44. *Viết chương trình cho phép nhận vào một chương trình viết bằng ngôn ngữ MINI PASCAL chứa trong một file text và thực hiện chương trình này.

Ngôn ngữ MINI PASCAL là ngôn ngữ PASCAL thu gọn, chỉ gồm:

- Kiểu dữ liệu INTEGER, REAL
- Các toán tử và hàm toán học như trong bài tập 17
- Các câu lệnh gán, IF THEN ESLE, FOR TO DO, WRITE
- Các từ khóa PROGRAM, VAR, BEGIN, END
- Không có chương trình con.

CHƯƠNG 4

CẤU TRÚC CÂY

Mục tiêu:

- Giới thiệu khái niệm cấu trúc cây.
- Cấu trúc dữ liệu cây nhị phân tìm kiếm: tổ chức, các thuật toán, ứng dụng.
- Giới thiệu cấu trúc dữ liệu cây nhị phân tìm kiếm cân bằng.

I. CẤU TRÚC CÂY

Định nghĩa 1: *Cây là một tập hợp T các phần tử (gọi là nút của cây) trong đó có một nút đặc biệt được gọi là gốc, các nút còn lại được chia thành những tập rời nhau T_1, T_2 , ... , T_n theo quan hệ phân cấp trong đó T_i cũng là một cây. Mỗi nút ở cấp i sẽ quản lý một số nút ở cấp i+1. Quan hệ này người ta còn gọi là quan hệ cha-con.*

Định nghĩa 2: *Cấu trúc cây với kiểu cơ sở T là một nút cấu trúc rỗng được gọi là cây rỗng (NULL). Một nút mà thông tin chính của nó có kiểu T, nó liên kết với một số hữu hạn các cấu trúc cây khác cũng có kiểu cơ sở T. Các cấu trúc này được gọi là những cây con của cây đang xét.*

1. Một số khái niệm cơ bản

- *Bậc của một nút*: là số cây con của nút đó .
- *Bậc của một cây:* là bậc lớn nhất của các nút trong cây (số cây con tối đa của một nút thuộc cây). Cây có bậc n thì gọi là cây n-phân.
- *Nút gốc:* là nút không có nút cha.
- *Nút lá:* là nút có bậc bằng 0 .
- *Nút nhánh:* là nút có bậc khác 0 và không phải là gốc .
- *Mức của một nút:*
  - Mức (gốc (T)) = 0.
  - Gọi $T_1, T_2, T_3, \ldots, T_n$ là các cây con của T_0

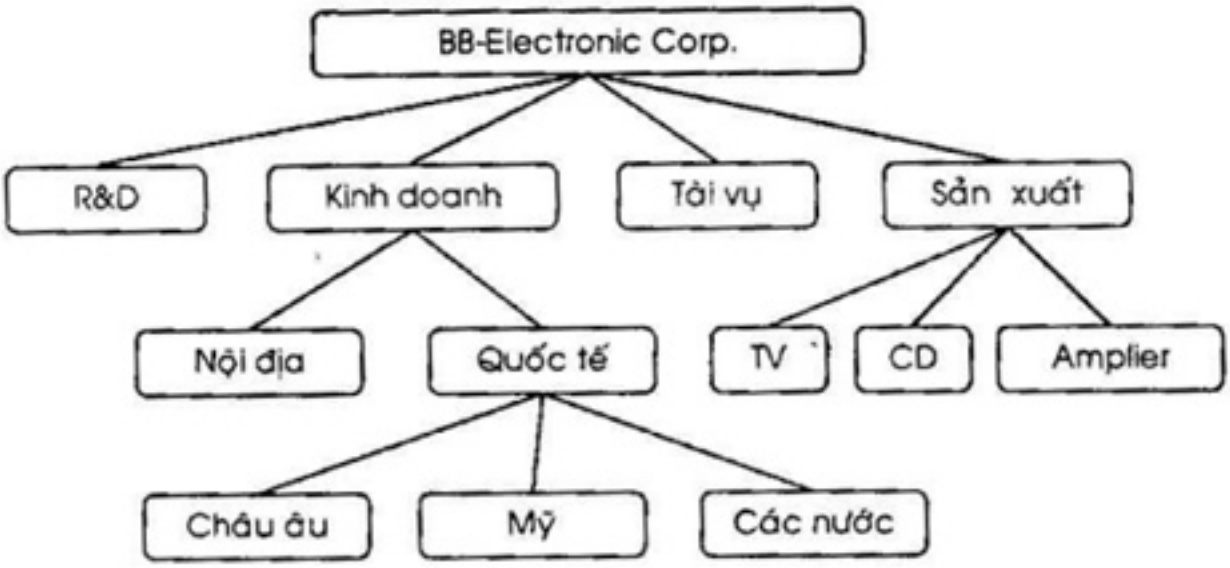
    Mức (T_1) = Mức (T_2) = ... = Mức (T_n) = Mức (T_0) + 1.
- *Độ dài đường đi từ gốc đến nút x:* là số nhánh cần đi qua kể từ gốc đến x.
- *Độ dài đường đi tổng của cây :*

$$P_T = \sum_{X \in T} P_X$$

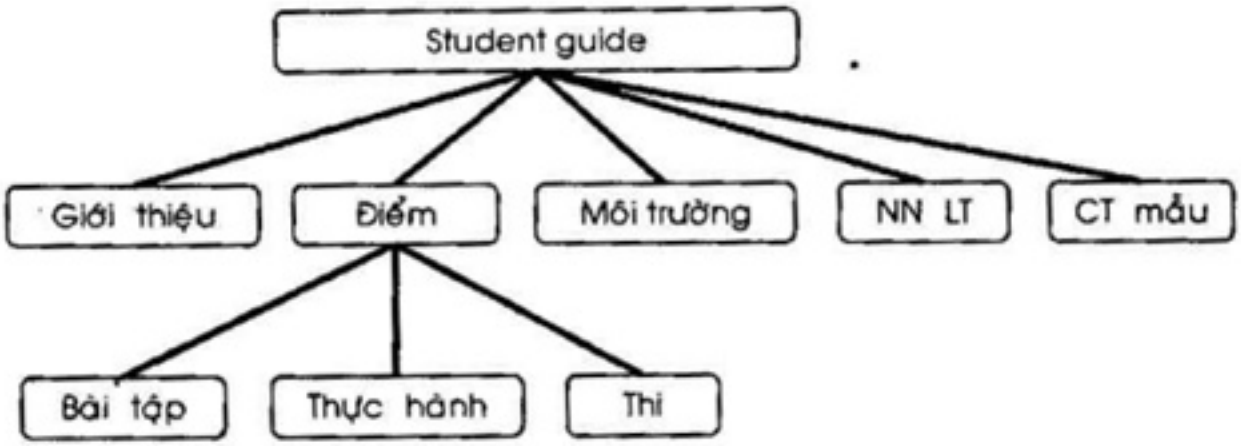
  trong đó P_x là độ dài đường đi từ gốc đến X.
- *Độ dài đường đi trung bình :* $P_I = P_T/n$ (n là số nút trên cây T).
- *Rừng cây:* là tập hợp nhiều cây trong đó thứ tự các cây là quan trọng.

2. Một số ví dụ về đối tượng các cấu trúc dạng cây

- Sơ đồ tổ chức của một công ty



- Mục lục một quyển sách



- Cấu trúc cây thư mục trong DOS/WIN
- Cấu trúc thư viện, ...

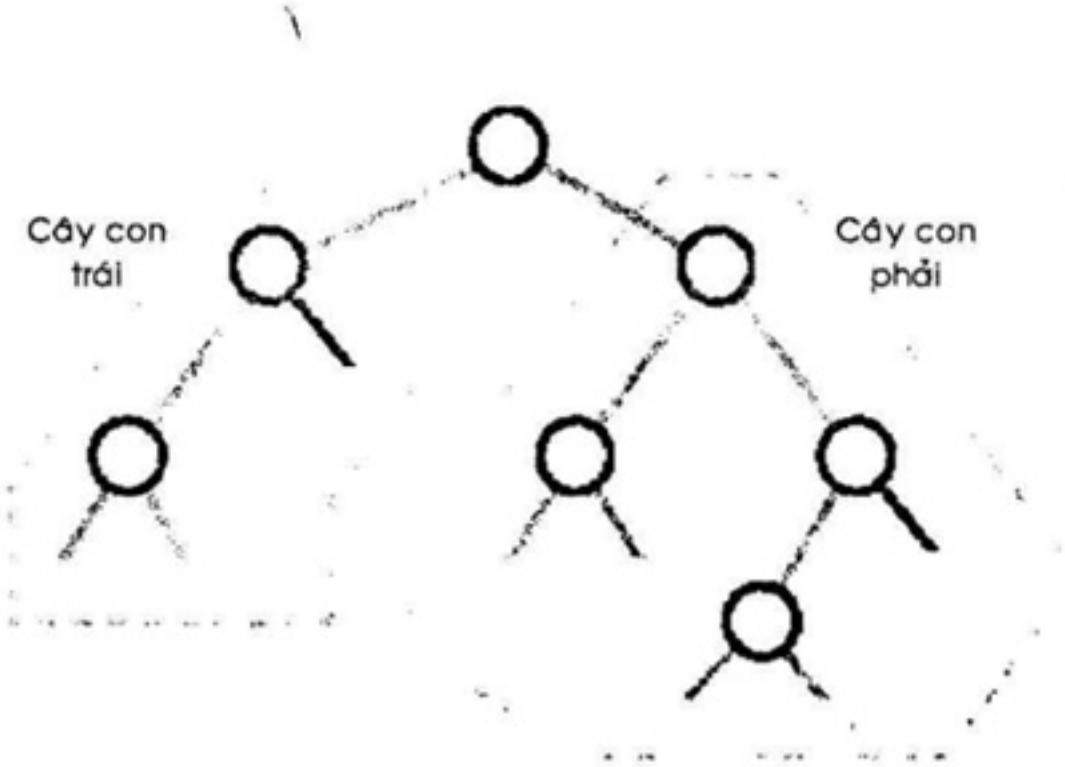
NHẬN XÉT

- Trong cấu trúc cây không tồn tại chu trình;
- Tổ chức một cấu trúc cây cho phép truy cập nhanh đến các phần tử của nó.

II. CÂY NHỊ PHÂN

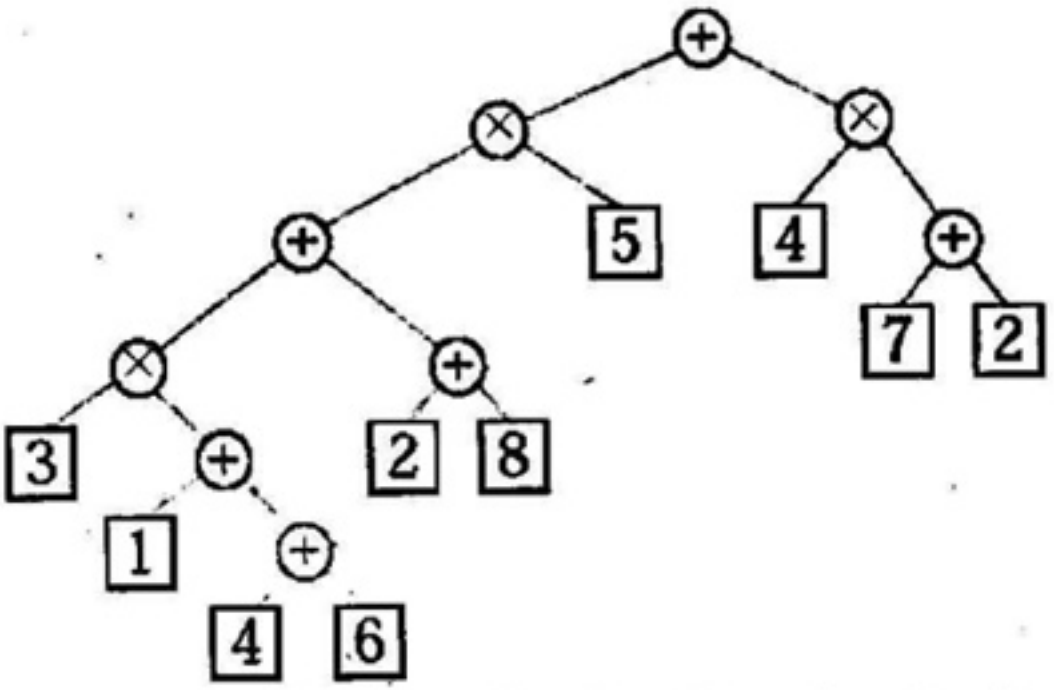
Định nghĩa: *Cây nhị phân là cây mà mỗi nút có tối đa hai cây con*

Trong thực tế thường gặp các cấu trúc có dạng cây nhị phân. Một cây tổng quát có thể biểu diễn thông qua cây nhị phân.



Hình ảnh một cây nhị phân

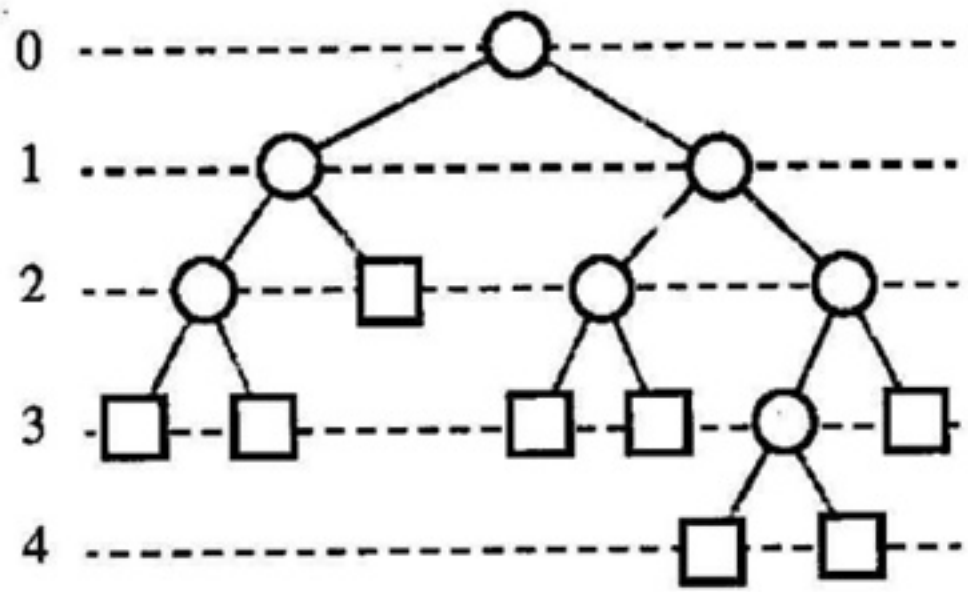
Cây nhị phân có thể ứng dụng trong nhiều bài toán thông dụng. Ví dụ dưới đây cho ta hình ảnh của một biểu thức toán học:



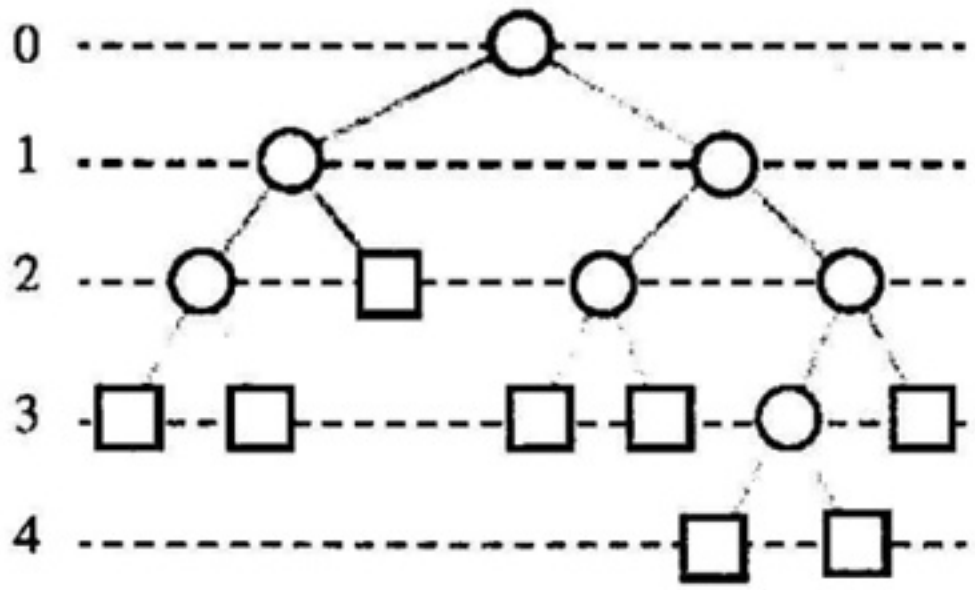
3 × 1 + 4 + 6 + 2 + 8 × 5 + 4 × 7 + 2

1. Một số tính chất của cây nhị phân

- Số nút nằm ở mức $I \leq 2^I$.
- Số nút lá $\leq 2^{h-1}$, với h là chiều cao của cây.
- Chiều cao của cây $h \geq \log_2$(số nút trong cây).
- Số nút trong cây $\leq 2^{h-1}$



- **Số nút trong cây ≤ 2^{h-1}**



2. Biểu diễn cây nhị phân T

Cây nhị phân là một cấu trúc bao gồm các phần tử (nút) được kết nối với nhau theo quan hệ "cha-con" với mỗi cha có tối đa 2 con. Để biểu diễn cây nhị phân ta chọn phương pháp cấp phát liên kết. Ứng với một nút, ta dùng một biến động lưu trữ các thông tin:

- Thông tin lưu trữ tại nút;
- Địa chỉ nút gốc của cây con trái trong bộ nhớ;
- Địa chỉ nút gốc của cây con phải trong bộ nhớ.

Khai báo tương ứng trong ngôn ngữ C có thể như sau:

```
typedef struct tagTNODE
{
    Data   Key;//Data là kiểu dữ liệu ứng với thông tin lưu tại nút
    struct tagNODE *pLeft, *pRight;
}TNODE;
typedef TNODE  *TREE;
```

Do tính chất mềm dẻo của cách biểu diễn bằng cấp phát liên kết, phương pháp này được dùng chủ yếu trong biểu diễn cây nhị phân. Từ đây trở đi, khi nói về cây nhị phân, chúng ta sẽ dùng

phương pháp biểu diễn này.

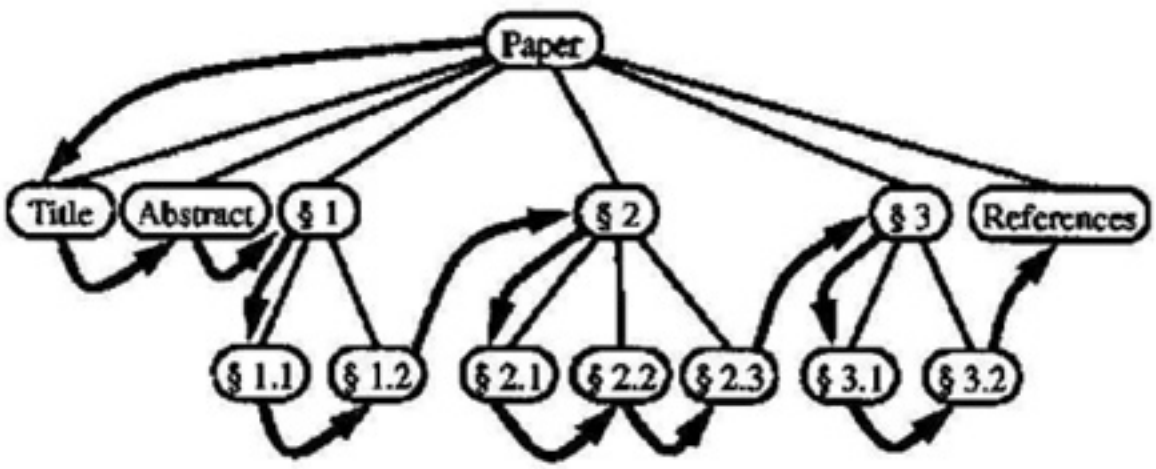
3. Duyệt cây nhị phân

Nếu như khi khảo sát cấu trúc dữ liệu dạng danh sách liên kết ta không quan tâm nhiều đến bài toán duyệt qua tất cả các phần tử của chúng thì bài toán duyệt cây hết sức quan trọng. Nó là cốt lõi của nhiều thao tác quan trong khác trên cây. Do cây nhị phân là một cấu trúc dữ liệu phi tuyến nên bài toán duyệt cây là bài toán không tầm thường.

Có nhiều kiểu duyệt cây khác nhau, và chúng cũng có những ứng dụng khác nhau. Đối với cây nhị phân, do cấu trúc đệ qui của nó, việc duyệt cây tiếp cận theo kiểu đệ qui là hợp lý và đơn giản nhất. Sau đây chúng ta sẽ xem xét một số kiểu duyệt thông dụng.

Có ba kiểu duyệt chính có thể áp dụng trên cây nhị phân: duyệt theo thứ tự trước (NLR), thứ tự giữa (LNR) và thứ tự sau (LRN). Tên của ba kiểu duyệt này được đặt dựa trên trình tự của việc thăm nút gốc so với việc thăm hai cây con.

Duyệt theo thứ tự trước (Node-Left-Right)



Kiểu duyệt này trước tiên thăm nút gốc sau đó thăm các nút của cây con trái rồi đến cây con phải. Thủ tục duyệt có thể trình bày đơn giản như sau:

```
void  NLR(TREE Root)
{
   if (Root != NULL)
   {
      <Xử lý Root>;    //Xử lý tương ứng theo nhu cầu
      NLR(Root->pLeft);
      NLR(Root->pRight);
   }
}
```

Duyệt theo thứ tự giữa (Left- Node-Right)

Kiểu duyệt này trước tiên thăm các nút của cây con trái sau đó thăm nút gốc rồi đến cây con phải. Thủ tục duyệt có thể trình bày đơn giản như sau:

```
void  LNR(TREE Root)
{
   if (Root != NULL)
   {
      NLR(Root->Left);
      <Xử lý Root>;          //Xử lý tương ứng theo nhu cầu
      NLR(Root->Right);
   }
}
```

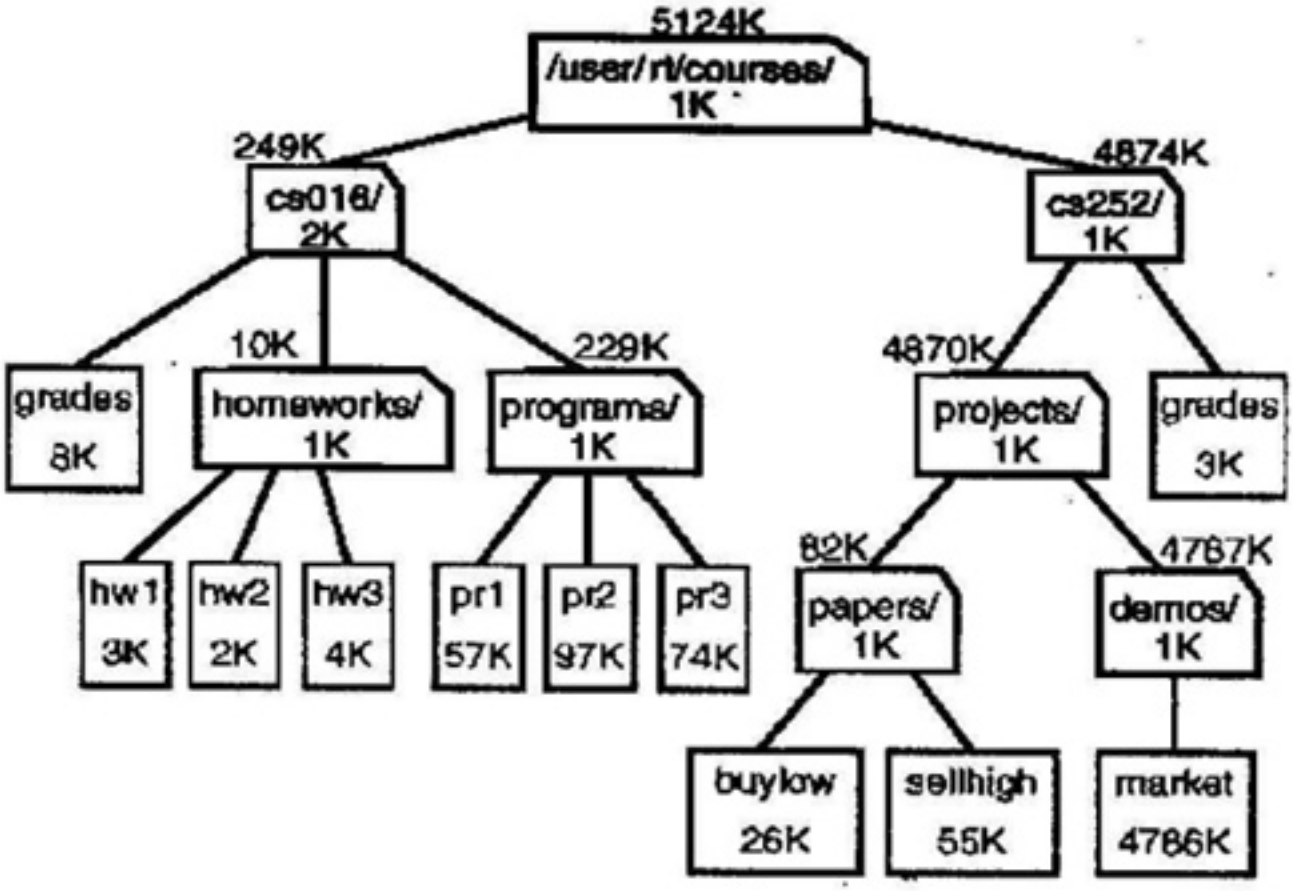
Duyệt theo thứ tự sau (Left-Right-Node)

Kiểu duyệt này trước tiên thăm các nút của cây con trái sau đó thăm đến cây con phải rồi cuối cùng mới thăm nút gốc. Thủ tục duyệt có thể trình bày đơn giản như sau:

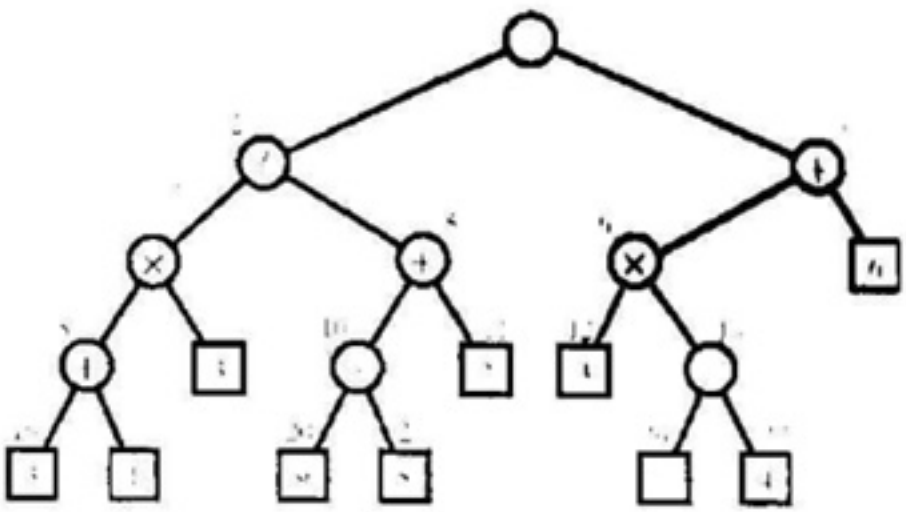
```
void  LRN(TREE Root)
{
   if (Root != NULL)
   {
```

```
        NLR(Root->Left);
        NLR(Root->Right);
        <Xử lý Root>;    //Xử lý tương ứng theo nhu cầu
    }
}
```

Một ví dụ quen thuộc trong tin học về ứng dụng của duyệt theo thứ tự sau là việc xác định tổng kích thước của một thư mục trên đĩa như hình sau:



Một ứng dụng quan trọng khác của phép duyệt cây theo thứ tự sau là việc tính toán giá trị của biểu thức dựa trên cây biểu thức như hình dưới:



(3 + 1)×3/(9 − 5 + 2) − (3×(7 − 4) + 6) = −13

4. Biểu diễn cây tổng quát bằng cây nhị phân

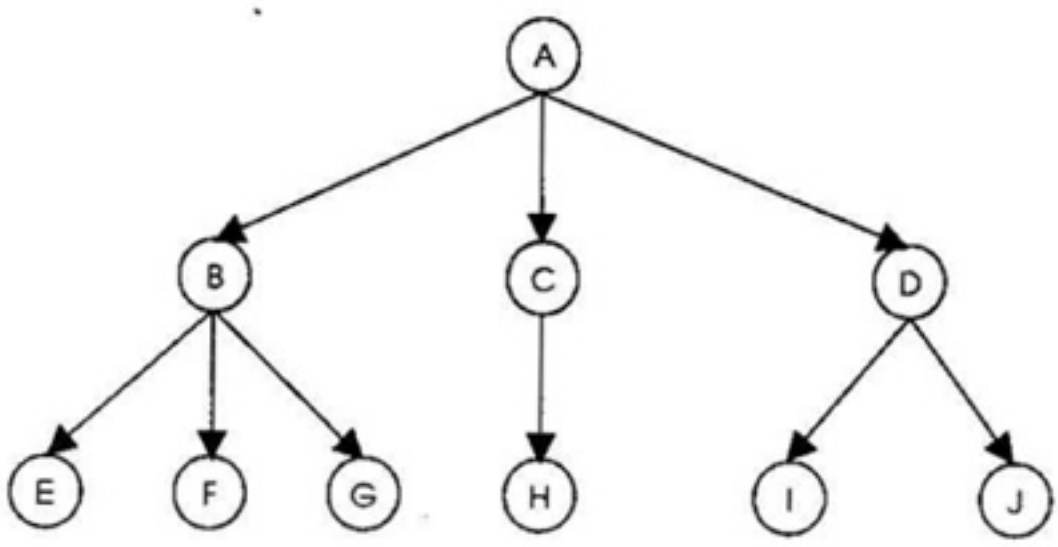
Nhược điểm của các cấu trúc cây tổng quát là bậc của các nút trên cây có thể dao động trong một biên độ lớn ⇒ việc biểu diễn gặp nhiều khó khăn và lãng phí. Hơn nữa, việc xây dựng các thao tác trên cây tổng quát phức tạp hơn trên cây nhị phân nhiều. Vì vậy, thường nếu không quá cần thiết phải sử dụng cây tổng quát, người ta chuyển cây tổng quát thành cây nhị phân.

Ta có thể biến đổi một cây bất kỳ thành một cây nhị phân theo qui tắc sau:

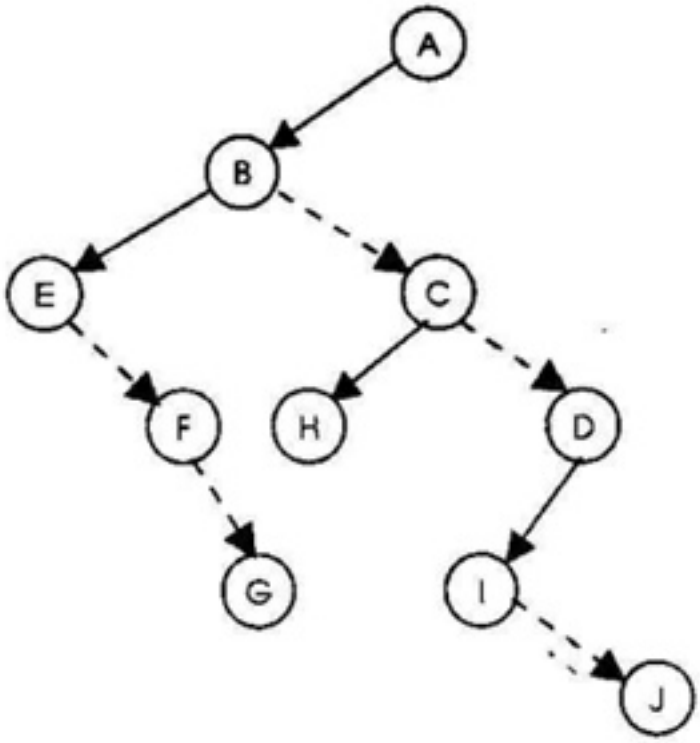
- **Giữ lại nút con trái nhất làm nút con trái.**
- **Các nút con còn lại chuyển thành nút con phải.**
- **Như vậy, trong cây nhị phân mới, con trái thể hiện quan hệ cha con và con phải thể hiện quan hệ anh em trong cây tổng quát ban đầu.**

Ta có thể xem ví dụ dưới đây để thấy rõ hơn qui trình.

Giả sử có cây tổng quát như hình bên dưới:



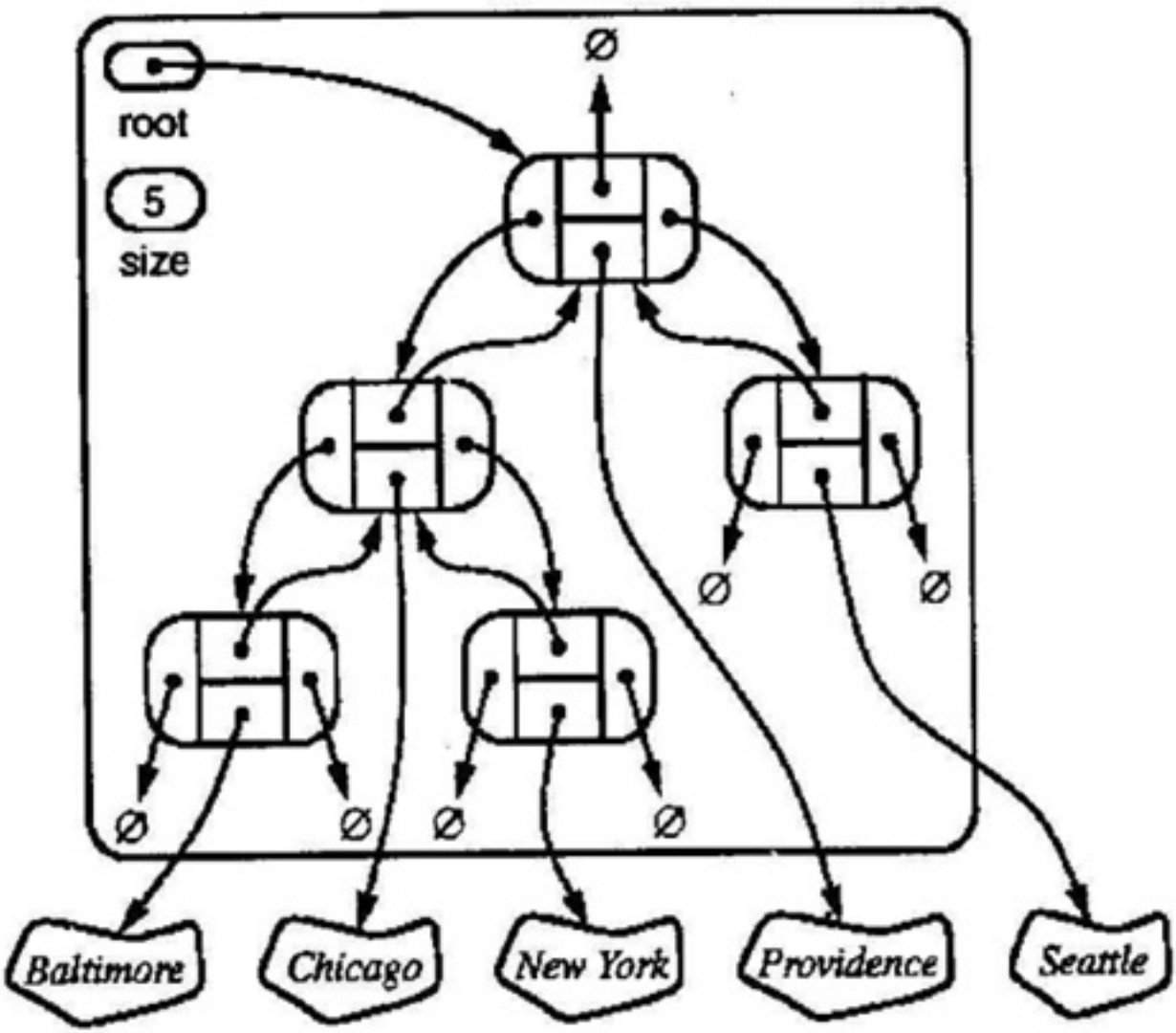
Cây nhị phân tương ứng sẽ như sau:



5. Một cách biểu diễn cây nhị phân khác

Đôi khi, khi định nghĩa cây nhị phân, người ta quan tâm đến cả quan hệ hai chiều cha con chứ không chỉ một chiều như định nghĩa ở phần trên. Lúc đó, cấu trúc cây nhị phân có thể định nghĩa lại như sau:

```
typedef struct tagTNode
{
   DataType            Key;
   struct tagTNode*    pParent;
   struct tagTNode*    pLeft;
   struct tagTNode*    pRight;
}TNODE;

typedef TNODE  *TREE;
```

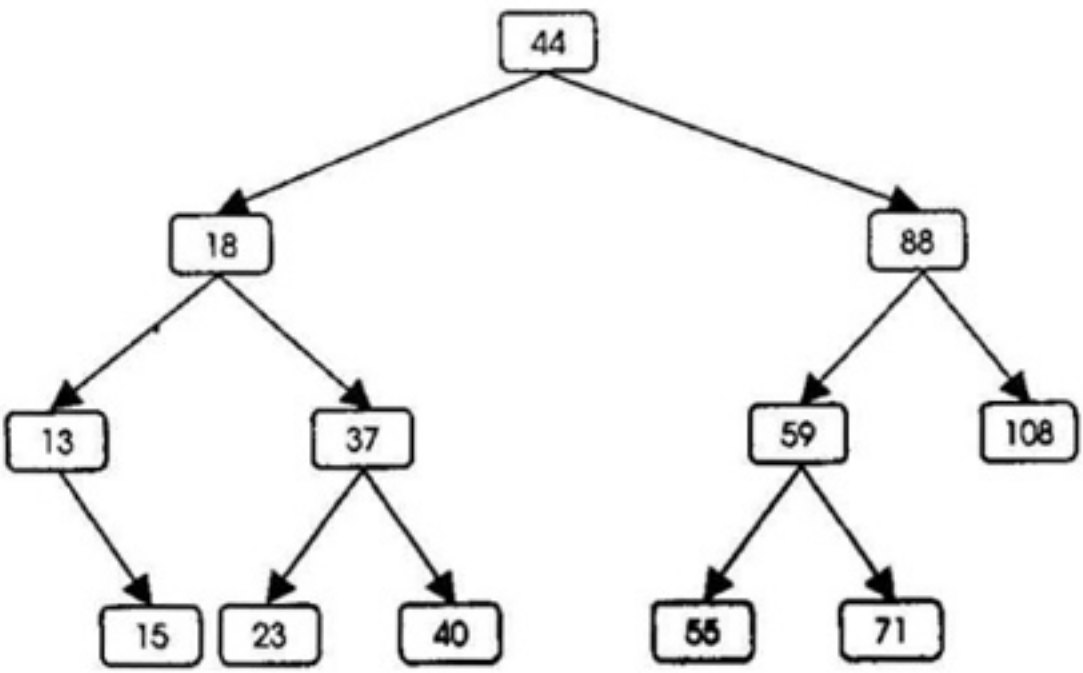


Trong chương 3, chúng ta đã làm quen với một số cấu trúc dữ liệu động. Các cấu trúc này có sự mềm dẻo nhưng lại bị hạn chế trong việc tìm kiếm thông tin trên chúng (chỉ có thể tìm kiếm tuần tự). Nhu cầu tìm kiếm là rất quan trọng. Vì lý do này, người ta đã đưa ra cấu trúc cây để thỏa mãn nhu cầu trên. Tuy nhiên, nếu chỉ với cấu trúc cây nhị phân đã định nghĩa ở trên, việc tìm kiếm còn rất mơ hồ. Cần có thêm một số ràng buộc để cấu trúc cây trở nên chặt chẽ, dễ dùng hơn. Một cấu trúc như vậy chính là **cây nhị phân tìm kiếm.**

III. CÂY NHỊ PHÂN TÌM KIẾM

Định nghĩa: *Cây nhị phân tìm kiếm (CNPTK) là cây nhị phân trong đó tại mỗi nút, khóa của nút đang xét lớn hơn khóa của tất cả các nút thuộc cây con trái và nhỏ hơn khóa của tất cả các nút thuộc cây con phải.*

Dưới đây là một ví dụ về cây nhị phân tìm kiếm:



Nhờ ràng buộc về khóa trên CNPTK, việc tìm kiếm trở nên có định hướng. Hơn nữa, do cấu trúc cây việc tìm kiếm trở nên nhanh đáng kể. Nếu số nút trên cây là N thì chi phí tìm kiếm trung bình chỉ khoảng $\log_2 N$.

Trong thực tế, khi xét đến cây nhị phân chủ yếu người ta xét CNPTK.

1. Các thao tác trên cây nhị phân tìm kiếm

Duyệt cây

Thao tác duyệt cây trên cây nhị phân tìm kiếm hoàn toàn giống như trên cây nhị phân. Chỉ có một lưu ý nhỏ là khi duyệt theo thứ tự giữa, trình tự các nút duyệt qua sẽ cho ta một dãy các nút theo thứ tự tăng dần của khóa.

Tìm một phần tử x trong cây

```
TNODE* searchNode(TREE T, Data X)
{
   if(T) {
      if(T->Key == X)return T;
      if(T->Key > X)
      return searchNode(T->pLeft, X);
      else
      return searchNode(T->pRight, X);
   }
   return NULL;
}
```

Ta có thể xây dựng một hàm tìm kiếm tương đương không đệ qui như sau:

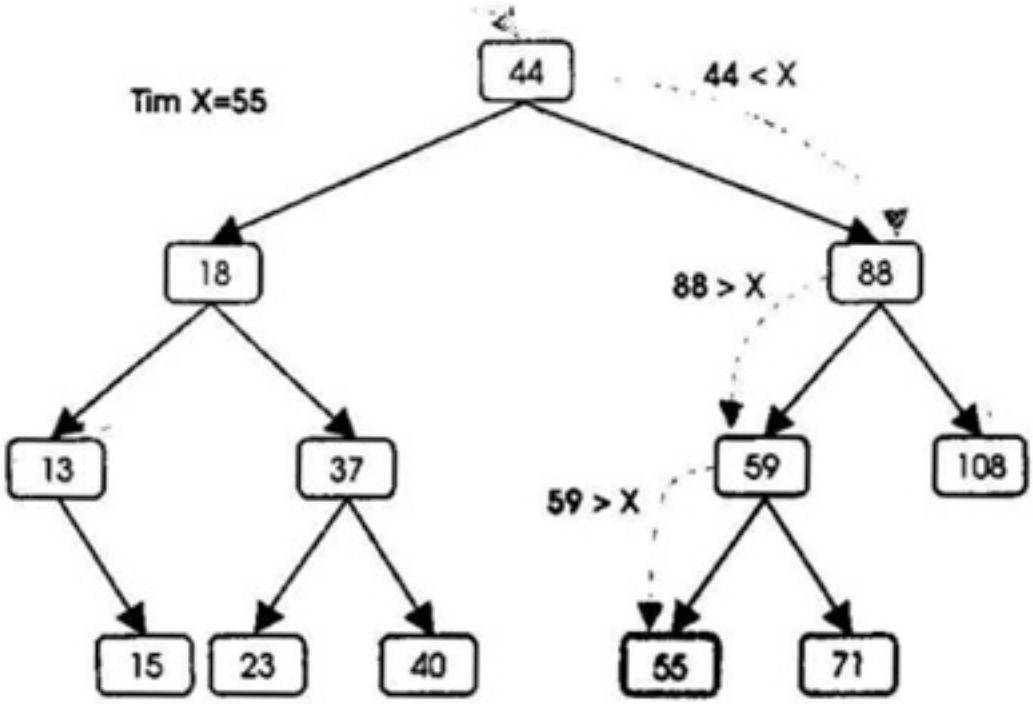
```
TNODE * searchNode(TREE Root,  Data x)
{  NODE *p = Root;

   while (p != NULL)
   {
      if(x == p->Key) return p;
      else
         if(x < p->Key) p = p->pLeft;
         else  p = p->pRight;
   }
   return NULL;
```

}

Dễ dàng thấy rằng số lần so sánh tối đa phải thực hiện để tìm phần tử X là h, với h là chiều cao của cây. Như vậy thao tác tìm kiếm trên CNPTK có n nút tốn chi phí trung bình khoảng $O(\log_2 n)$

.

Ví dụ



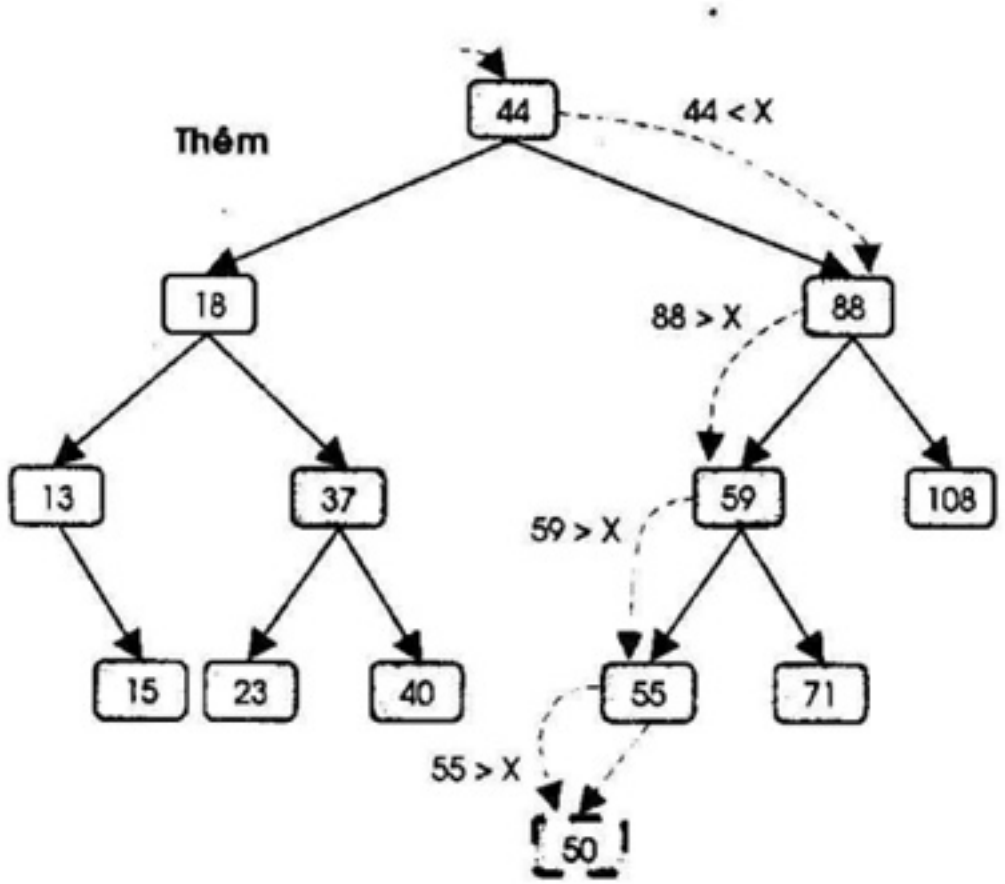
Thêm một phần tử x vào cây

Việc thêm một phần tử X vào cây phải bảo đảm điều kiện ràng buộc của CNPTK. Ta có thể thêm vào nhiều chỗ khác nhau trên cây, nhưng nếu thêm vào một nút lá sẽ là tiện lợi nhất do ta có thể thực hiên quá trình tương tự thao tác tìm kiếm. Khi chấm dứt quá trình tìm kiếm cũng chính là lúc tìm được chỗ cần thêm.

Hàm insert trả về giá trị –1, 0, 1 khi không đủ bộ nhớ, gặp nút cũ hay thành công:

```
int insertNode(TREE &T, Data X)
{
   if(T) {
      ir(T->Key == X) return 0; //đã có
      if(T->Key > X)
      return insertNode(T->pLeft, X);
      else
      return insertNode(T->pRight, X);
   }
   T  = new TNode;
   if(T == NULL)   return -1; //thiếu bộ nhớ
   T->Key= X;
   T->pLeft =T->pRight = NULL;
   return 1; //thêm vào thành công
}
```

Ví dụ



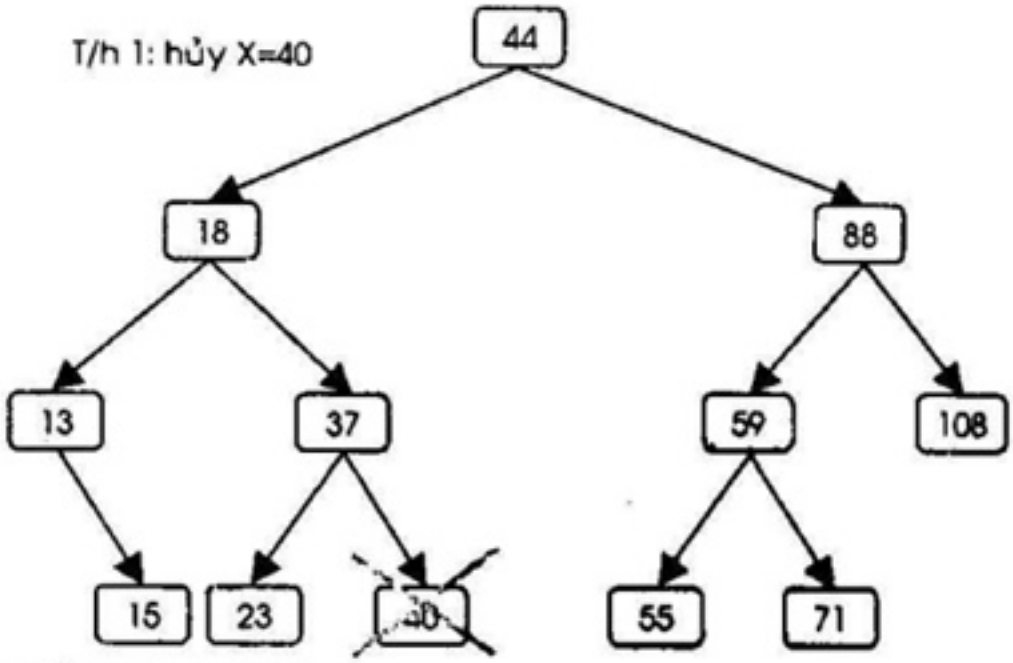
Hủy một phần tử có khóa x

- Việc hủy một phần tử X ra khỏi cây phải bảo đảm điều kiện ràng buộc của CNPTK.

Có ba trường hợp khi hủy nút X có thể xảy ra:

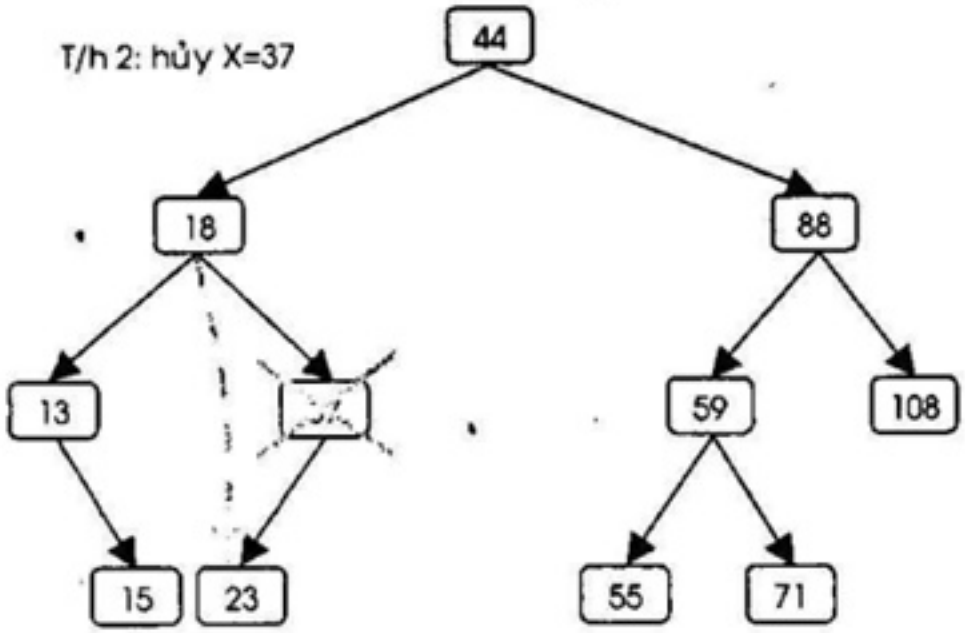
- **X là nút lá.**
- **X chỉ có một con (trái hoặc phải).**
- **X có đủ cả hai con**

Trường hợp thứ nhất: chỉ đơn giản hủy X vì nó không móc



nối đến phần tử nào khác.

Trường hợp thứ hai: trước khi hủy X ta móc nối cha của X với con duy nhất của nó.



Trường hợp cuối cùng: ta không thể hủy trực tiếp do X có đủ hai con ⇒ Ta sẽ hủy gián tiếp. Thay vì hủy X, ta sẽ tìm một phần tử thế mạng Y. Phần tử này có tối đa một con. Thông tin lưu tại Y sẽ được chuyển lên lưu tại X. Sau đó, nút bị hủy thật sự sẽ là Y giống như hai trường hợp đầu.

Vấn đề là phải chọn Y sao cho khi lưu Y vào vị trí của X, cây vẫn là CNPTK.

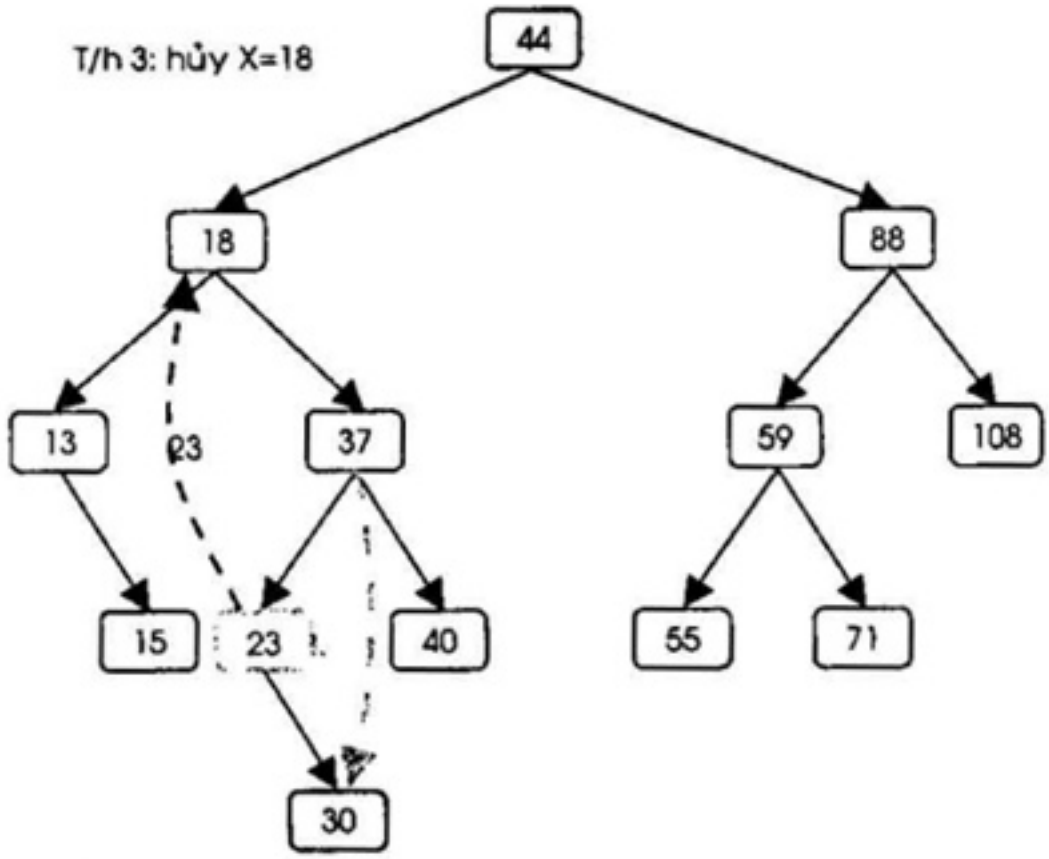
Có hai phần tử thỏa mãn yêu cầu:

- **Phần tử nhỏ nhất (trái nhất) trên cây con phải.**
- **Phần tử lớn nhất (phải nhất) trên cây con trái.**

Việc chọn lựa phần tử nào là phần tử thế mạng hoàn toàn phụ thuộc vào ý thích của người lập trình. Ở đây, chúng tôi sẽ chọn phần tử (phải nhất trên cây con trái) làm phần tử thế mạng.

Hãy xem ví dụ dưới đây để hiểu rõ hơn:

Sau khi hủy phần tử X =18 ra khỏi cây tình trạng của cây sẽ như trong hình dưới đây (phần tử 23 là phần tử thế mạng):



Hàm delNode trả về giá trị 1, 0 khi hủy thành công hoặc không có X trong cây:

```
int delNode(TREE &T, Data X)
{
   if(T==NULL) return 0;
   if(T->Key > X)
   return delNode (T->pLeft, X);
   if(T->Key < X)
   return delNode (T->pRight, X);
   else  { //T->Key == X
      TNode*p = T;
      if(T->pLeft == NULL)
      T  = T->pRight;
      else if(T->pRight == NULL)
            T  = T->pLeft;
            else  {  //T có cả 2 con
               TNode*q = T->pRight;
               searchStandFor(p, q);
```

```
            }
        delete p;
    }
}
```

trong đó, hàm searchStandFor được viết như sau:

```
//Tìm phần tử thế mạng cho nút p
void searchStandFor(TREE &p, TREE &q)
{
      if(q->pLeft)
            searchStandFor(p, q->pLeft);
      else  {
            p->Key   = q->Key;
            p  = q;
            q  = q->pRight;
      }
}
```

Tạo một cây CNPTK

Ta có thể tạo một cây nhị phân tìm kiếm bằng cách lặp lại quá trình thêm một phần tử vào một cây rỗng.

Hủy toàn bộ CNPK

Việc toàn bộ cây có thể được thực hiện thông qua thao tác duyệt cây theo thứ tự sau. Nghĩa là ta sẽ hủy cây con trái, cây con phải rồi mới hủy nút gốc.

```
void removeTree(TREE &T)
{
   if(T) {
      removeTree(T->pLeft);
      removeTree(T->pRight);
      delete(T);
   }
}
```

NHẬN XÉT

- Tất cả các thao tác searchNode, insertNode, delNode trên CNPTK đều có độ phức tạp trung bình O(h), với h là chiều cao của cây.

- Trong trong trường hợp tốt nhất, CNPTK có n nút sẽ có độ cao h = $\log_2(n)$. Chi phí tìm kiếm khi đó sẽ tương đương tìm kiếm nhị phân trên mảng có thứ tự.

- Tuy nhiên, trong trường hợp xấu nhất, cây có thể bị suy biến thành một DSLK (khi mà mỗi nút đều chỉ có một con trừ nút lá). Lúc đó các thao tác trên sẽ có độ phức tạp O(n). Vì vậy cần có cải tiến cấu trúc của CNPTK để đạt được chi phí cho các thao tác là $\log_2(n)$.

IV. CÂY NHỊ PHÂN CÂN BẰNG (AVL TREE)

1. Cây nhị phân cân bằng hoàn toàn

Định nghĩa: *Cây nhị phân cân bằng hoàn toàn là cây nhị phân tìm kiếm mà tại mỗi nút của nó, số nút của cây con trái chênh lệch không quá một so với số nút của cây con phải.*

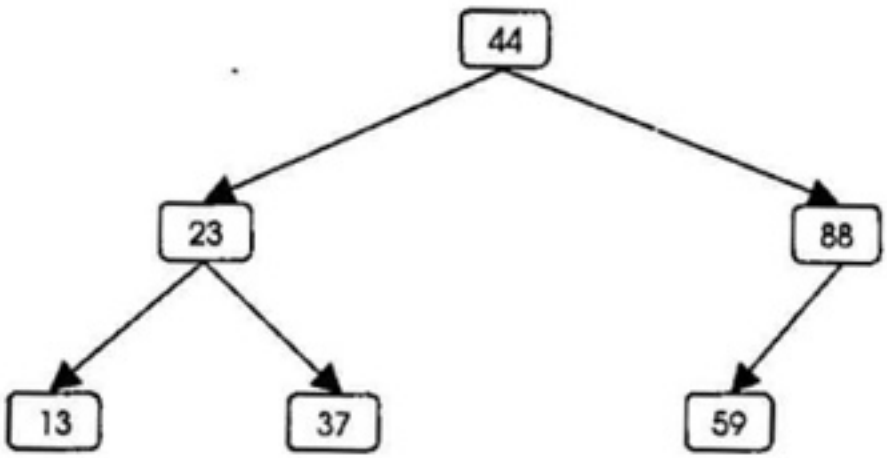
Một cây rất khó đạt được trạng thái cân bằng hoàn toàn và cũng rất dễ mất cân bằng vì khi thêm hay hủy các nút trên cây có thể làm cây mất cân bằng (xác suất rất lớn), chi phí cân bằng lại cây lớn vì phải thao tác trên toàn bộ cây.

Tuy nhiên nếu cây cân đối thì việc tìm kiếm sẽ nhanh. Đối với cây cân bằng hoàn toàn, trong trường hợp xấu nhất ta chỉ phải tìm

qua $\log_2 n$ phần tử (n là số nút trên cây).

Sau đây là ví dụ một cây cân bằng hoàn toàn (CCBHT):

CCBHT có n nút có chiều cao $h \approx \log_2 n$. Đây chính là lý do cho phép bảo đảm khả năng tìm kiếm nhanh trên CTDL này.



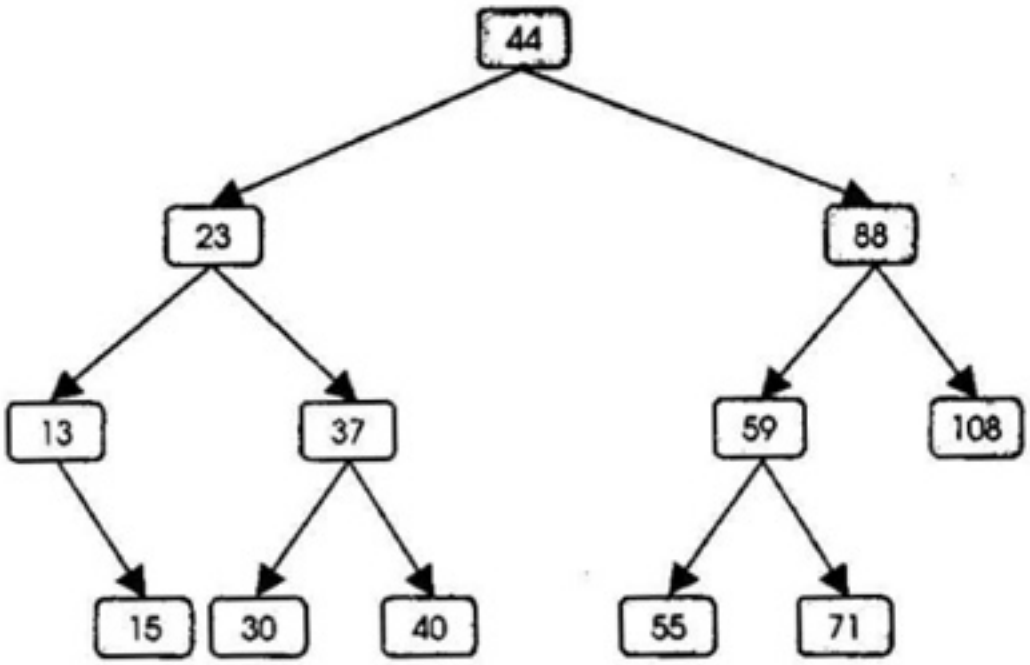
Do CCBHT là một cấu trúc kém ổn định nên trong thực tế không thể sử dụng. Nhưng ưu điểm của nó lại rất quan trọng. Vì vậy, cần đưa ra một CTDL khác có đặc tính giống CCBHT nhưng ổn định hơn.

Như vậy, cần tìm cách tổ chức một cây đạt trạng thái cân bằng yếu hơn và việc cân bằng lại chỉ xảy ra ở phạm vi cục bộ nhưng vẫn phải bảo đảm chi phí cho thao tác tìm kiếm đạt ở mức $O(\log_2 n)$.

2. Cây nhị phân cân bằng

Định nghĩa: *Cây nhị phân cân bằng là cây mà tại mỗi nút của nó độ cao của cây con trái và của cây con phải chênh lệch không quá một.*

Dưới đây là ví dụ cây cân bằng (lưu ý, cây này không phải là cây cân bằng hoàn toàn):



Dễ dàng thấy CCBHT là cây cân bằng. Điều ngược lại không đúng. Tuy nhiên cây cân bằng là CTDL ổn định hơn hẳn CCBHT.

Lịch sử cây cân bằng (AVL Tree)

AVL là tên viết tắt của các tác giả người Nga đã đưa ra định nghĩa của cây cân bằng Adelson-Velskii và Landis (1962). Vì lý do này, người ta gọi cây nhị phân cân bằng là cây AVL. Từ nay về sau, chúng ta sẽ dùng thuật ngữ cây AVL thay cho cây cân bằng.

Từ khi được giới thiệu, cây AVL đã nhanh chóng tìm thấy ứng dụng trong nhiều bài toán khác nhau. Vì vậy, nó mau chóng trở nên thịnh hành và thu hút nhiều nghiên cứu. Từ cây AVL, người ta đã phát triển thêm nhiều loại CTDL hữu dụng khác như cây đỏ-đen (Red-Black Tree), B-Tree, ...

Một vấn đề quan trọng, như đã đề cập đến ở phần trước, là ta pjải khẳng định cây AVL n nút phải có chiều cao khoảng $\log_2(n)$.

Để đánh giá chính xác về chiều cao của cây AVL, ta xét bài toán: cây AVL có chiều cao h sẽ phải có tối thiểu bao nhiêu nút ?

Gọi N(h) là số nút tối thiểu của cây AVL có chiều cao h.

Ta có N(0) = 0, N(1) = 1 và N(2) = 2.

Cây AVL tối thiểu có chiều cao h sẽ có một cây con AVL tối thiểu chiều cao h-1 và một cây con AVL tối thiểu chiều cao h-2. Như vậy:

$$N(h) = 1 + N(h-1) + N(h-2) \qquad (1)$$

Ta lại có: $N(h-1) > N(h-2)$

nên từ (1) suy ra:

$$N(h) > 2N(h-2)$$

$$N(h) > 2^2N(h-4)$$

...

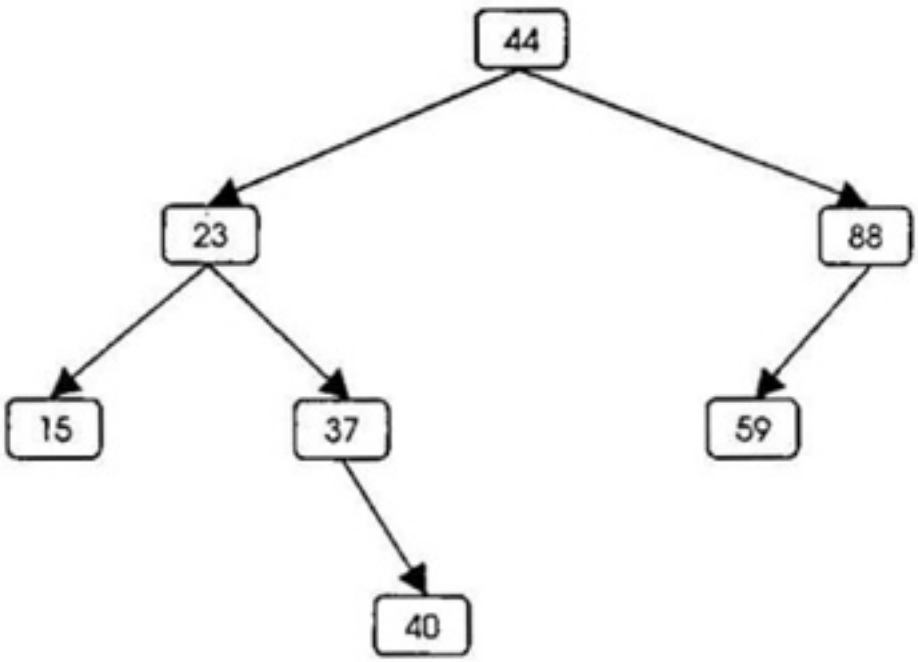
$$N(h) > 2^iN(h-2i)$$

$$\Rightarrow N(h) > 2^{h/2-1}$$

$$\Rightarrow h < 2\log_2(N(h)) + 2$$

Như vậy, cây AVL có chiều cao $O(\log_2(n))$.

Ví dụ: cây AVL tối thiểu có chiều cao h = 4



Chỉ số cân bằng của một nút

Định nghĩa: *Chỉ số cân bằng của một nút là hiệu của chiều cao cây con phải và cây con trái của nó.*

Đối với một cây cân bằng, chỉ số cân bằng (CSCB) của mỗi nút chỉ có thể mang một trong ba giá trị sau đây:

CSCB(p) = 0 ⇔ Độ cao cây trái (p) = Độ cao cây phải (p)

CSCB(p) = 1 ⇔ Độ cao cây trái (p) < Độ cao cây phải (p)

CSCB(p) =-1 ⇔ Độ cao cây trái (p) > Độ cao cây phải (p)

Để tiện trong trình bày, chúng ta sẽ ký hiệu như sau:

p->balFactor = CSCB(p);

Độ cao cây trái (p) ký hiệu là h_L

Độ cao cây phải(p) ký hiệu là h_R

Để khảo sát cây cân bằng, ta cần lưu thêm thông tin về chỉ số cân bằng tại mỗi nút. Lúc đó, cây cân bằng có thể được khai báo như sau:

```
typedef struct tagAVLNode  {
   char       balFactor; //Chỉ số cân bằng
   Data       key;
   struct tagAVLNode*   pLeft;
   struct tagAVLNode*   pRight;
}AVLNode;
typedef AVLNode              *AVLTree;
```

Để tiện cho việc trình bày, ta định nghĩa một số hằng số sau:

```
#define   LH -1 //Cây con trái cao hơn
#define   EH -0 //Hai cây con bằng nhau
#define   RH 1  //Cây con phải cao hơn
```

Ta nhận thấy trường hợp thêm hay hủy một phần tử trên cây có thể làm cây tăng hay giảm chiều cao, khi đó phải cân bằng lại cây. Việc cân bằng lại một cây sẽ phải thực hiện sao cho chỉ ảnh hưởng tối thiểu đến cây nhằm giảm thiểu chi phí cân bằng. Như đã nói ở trên, cây cân bằng cho phép việc cân bằng lại chỉ xảy ra trong giới hạn cục bộ nên chúng ta có thể thực hiện được mục tiêu vừa nêu.

Như vậy, ngoài các thao tác bình thường như trên CNPTK, các thao tác đặc trưng của cây AVL gồm:

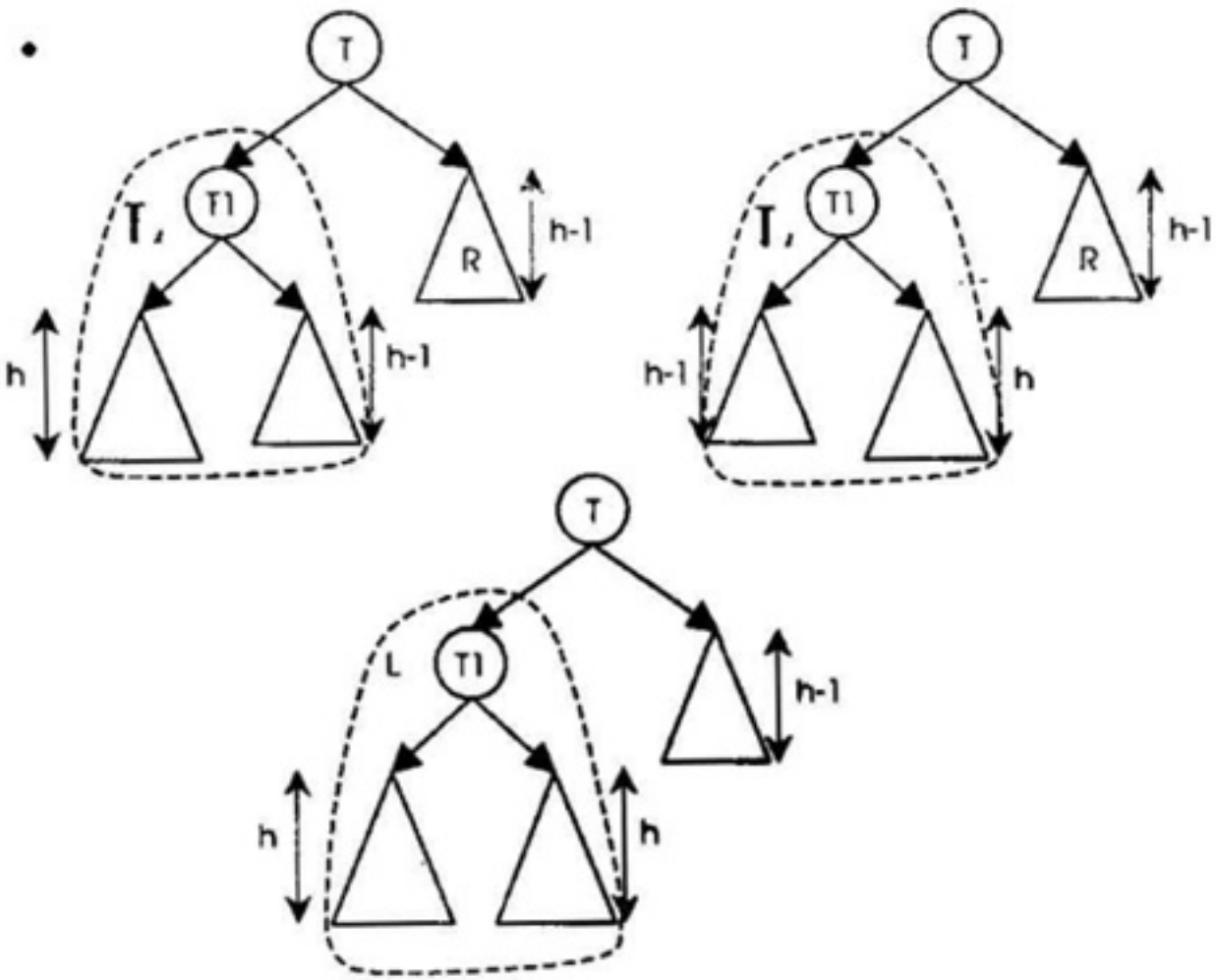
- **Thêm một phần tử vào cây AVL;**
- **Hủy một phần tử trên cây AVL;**
- **Cân bằng lại một cây vừa bị mất cân bằng.**

Các trường hợp mất cân bằng

Ta sẽ không khảo sát tính cân bằng của một cây nhị phân bất kỳ mà chỉ quan tâm đến các khả năng mất cân bằng xảy ra khi thêm hoặc hủy một nút trên cây AVL.

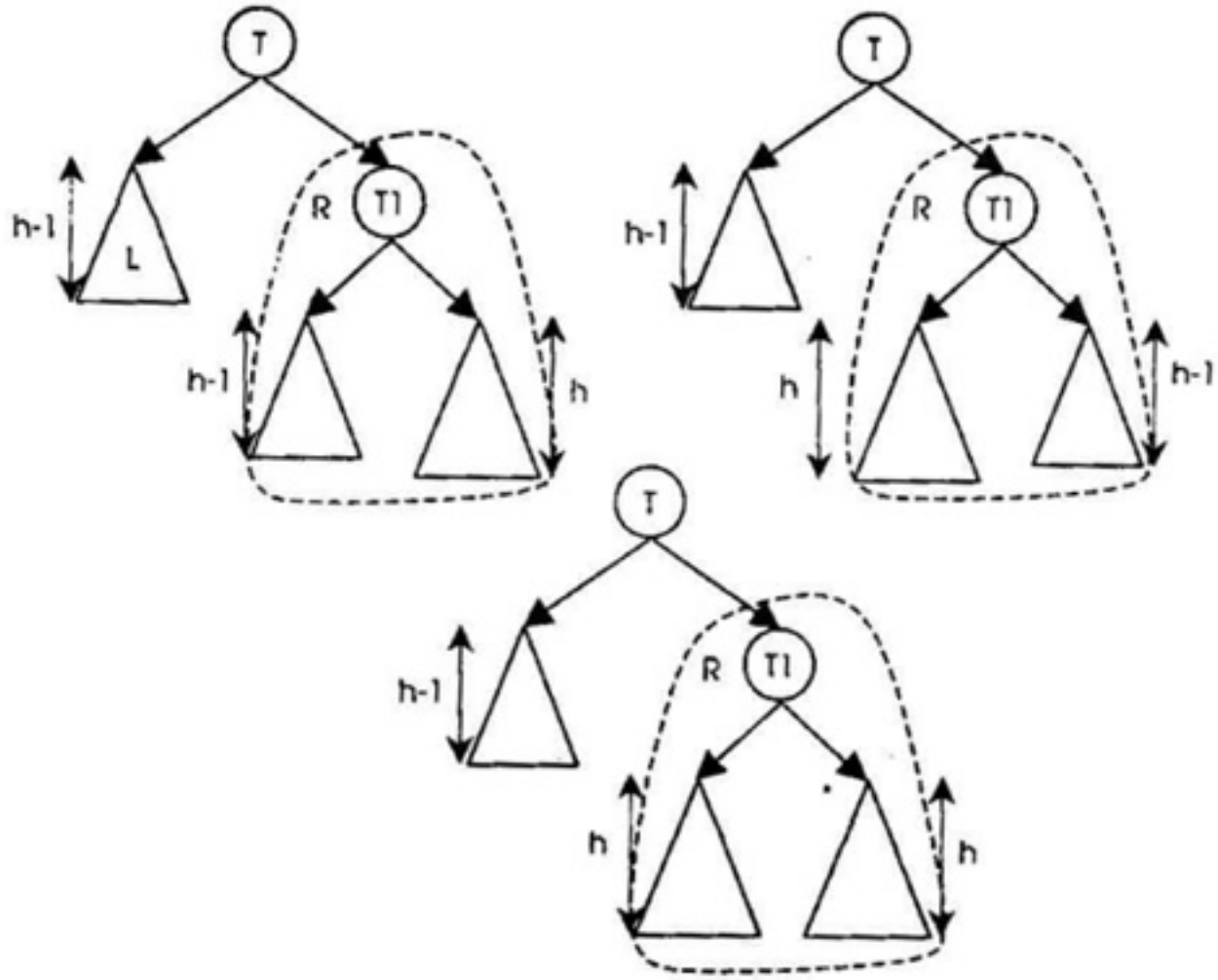
Như vậy, khi mất cân bằng, độ lệch chiều cao giữa hai cây con sẽ là **2**. Ta có sáu khả năng sau:

- **Trường hợp 1: cây T lệch về bên trái (có ba khả năng)**



Trường hợp 2: cây T lệch về bên phải

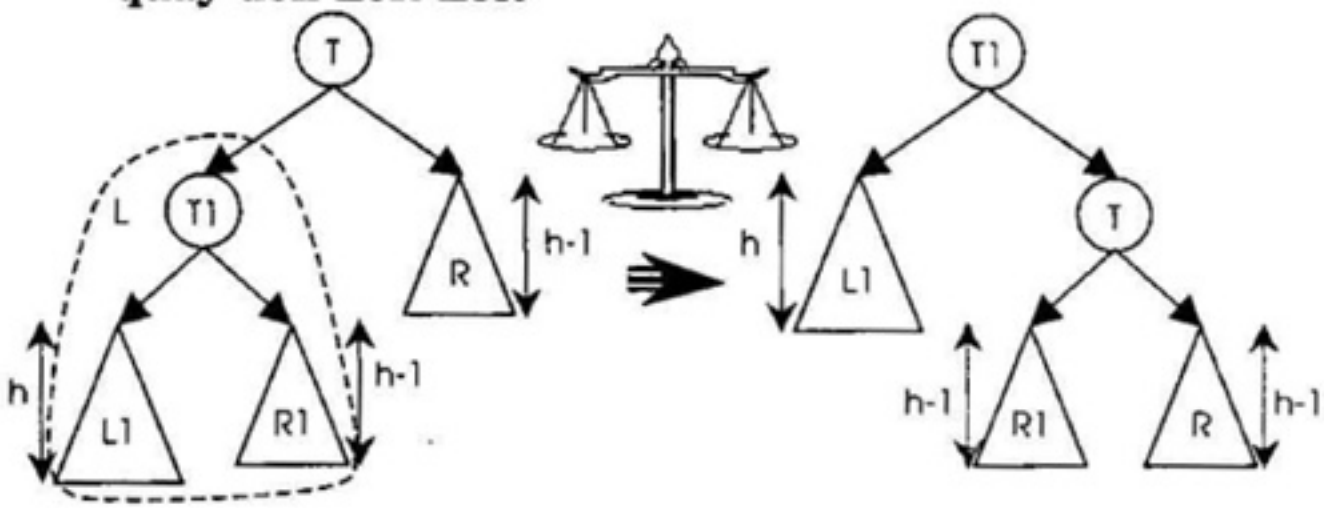
Ta có các khả năng sau:



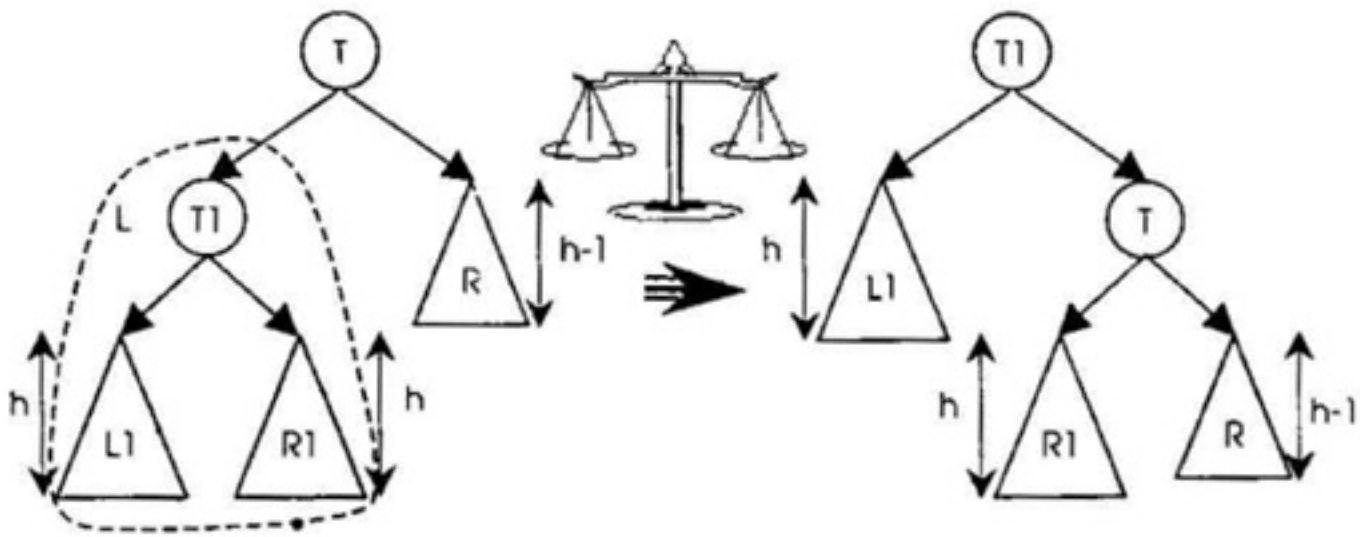
Ta có thể thấy rằng các trường hợp lệch về bên phải hoàn toàn đối xứng với các trường hợp lệch về bên trái. Vì vậy ta chỉ cần khảo sát trường hợp lệch về bên trái. Trong ba trường hợp lệch về bên trái, trường hợp T1 lệch phải là phức tạp nhất. Các trường hợp còn lại giải quyết rất đơn giản.

Sau đây, ta sẽ khảo sát và giải quyết từng trường hợp nêu trên.

- **T/h 1.1: cây T1 lệch về bên trái. Ta thực hiện phép quay đơn Left-Left**



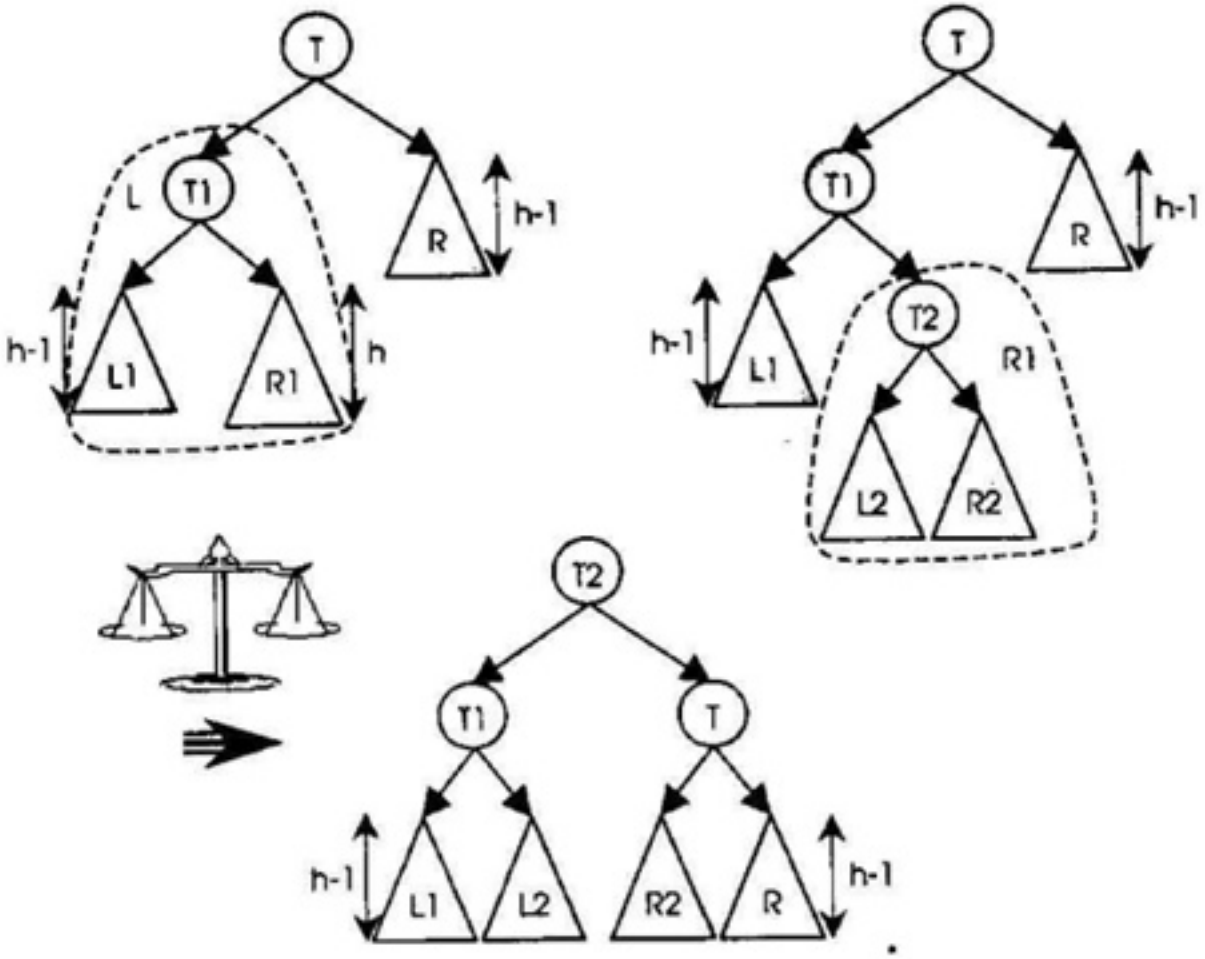
- **T/h 1.2: cây T1 không lệch. Ta thực hiện phép quay đơn Left-Left**



- **T/h 1.3: cây T1 lệch về bên phải. Ta thực hiện phép quay kép Left-Right**

Do T1 lệch về bên phải ta **không thể áp dụng phép** quay đơn đã áp dụng trong hai trường hợp trên vì khi đó cây T sẽ chuyển từ trạng thái mất cân bằng do lệch trái thành mất cân bằng do lệch phải ⇒ cần áp dụng cách khác.

Hình vẽ dưới đây minh họa phép quay kép áp dụng cho trường hợp này:



Lưu ý rằng, trước khi cân bằng cây T có chiều cao h + 2 trong cả ba trường hợp 1.1, 1.2 và 1.3. Sau khi cân bằng, trong hai trường hợp 1.1 và 1.3 cây có chiều cao h + 1; còn ở trường hợp 1.2 cây vẫn có chiều cao h + 2. Và trường hợp này cũng là trường hợp duy nhất sau khi cân bằng nút T cũ có chỉ số cân bằng ≠ 0.

Thao tác cân bằng lại trong tất cả các trường hợp đều có độ phức tạp O(1).

Với những xem xét trên, xét tương tự cho trường hợp cây T lệch về bên phải, ta có thể xây dựng hai hàm quay đơn và hai hàm quay kép sau:

```
//quay đơn Left-Left
void rotateLL(AVLTree &T)
{   AVLNode* T1 = T->pLeft;
```

```
    T->pLeft = T1->pRight;
    T1->pRight  = T;
    switch(T1->balFactor){
        case LH: T->balFactor = EH;
                 T1->balFactor  = EH; break;
        case EH: T->balFactor = LH;
                 T1->balFactor  = RH; break;
    }
    T  = T1;
}
```

//quay đơn Right-Right

```
void rotateRR(AVLTree &T)
{   AVLNode* T1 = T->pRight;

    T->pRight= T1->pLeft;
    T1->pLeft= T;
    switch(T1->balFactor){
           case RH: T->balFactor = EH;
                    T1->balFactor  = EH; break;
           case EH: T->balFactor = RH; break;
                    T1->balFactor  = LH; break;
    }
    T  = T1;
}
```

//quay kép Left-Right

```
void rotateLR(AVLTree &T)
{   AVLNode* T1 = T->pLeft;
    AVLNode* T2 = T1->pRight;
    T->pLeft = T2->pRight;
    T2->pRight  = T;
    T1->pRight  = T2->pLeft;
    T2->pLeft= T1;
    switch(T2->balFactor){
        case LH: T->balFactor = RH;
                 T1->balFactor = EH; break;
        case EH: T->balFactor = EH;
                 T1->balFactor = EH; break;
        case RH: T->balFactor = EH;
                 T1->balFactor = LH; break;
    }
    T2->balFactor  = EH;
    T  = T2;
}
```

```
//quay kép Right-Left
void rotateRL(AVLTree &T)
{  AVLNode* T1 = T->pRight;
   AVLNode* T2 = T1->pLeft;

   T->pRight= T2->pLeft;
   T2->pLeft= T;
   T1->pLeft= T2->pRight;
   T2->pRight  = T1;
   switch(T2->balFactor){
      case RH: T->balFactor = LH;
               T1->balFactor = EH; break;
      case EH: T->balFactor = EH;
               T1->balFactor = EH; break;
      case LH: T->balFactor = EH;
               T1->balFactor = RH; break;
   }
   T2->balFactor  = EH;
   T  = T2;
}
```

Để thuận tiện, ta xây dựng hai hàm cân bằng lại khi cây bị lệch trái hay lệch phải như sau:

```
//Cân bằng khi cây bị lệch về bên trái
int balanceLeft(AVLTree &T)
{  AVLNode* T1 = T->pLeft;

   switch(T1->balFactor)   {
      case LH: rotateLL(T); return 2;
      case EH: rotateLL(T); return 1;
      case RH: rotateLR(T); return 2;
   }
   return 0;
}
//Cân bằng khi cây bị lệch về bên phải
int balanceRight(AVLTree &T)
{  AVLNode* T1 = T->pRight;

   switch(T1->balFactor)   {
      case LH: rotateRL(T); return 2;
      case EH: rotateRR(T); return 1;
      case RH: rotateRR(T); return 2;
   }
```

```
    return 0;
}
```

Thêm một phần tử trên cây AVL

Việc thêm một phần tử vào cây AVL diễn ra tương tự như trên CNPTK. Tuy nhiên, sau khi thêm xong, nếu chiều cao của cây thay đổi, từ vị trí thêm vào, ta phải lần ngược lên gốc để kiểm tra xem có nút nào bị mất cân bằng không. Nếu có, ta phải cân bằng lại ở nút này.

Việc cân bằng lại chỉ cần thực hiện một lần tại nơi mất cân bằng. (*Tại sao ? Hd: chú ý những khả năng mất cân bằng có thể*)

Hàm insert trả về giá trị –1, 0, 1 khi không đủ bộ nhớ, gặp nút cũ hay thành công. Nếu sau khi thêm, chiều cao cây bị tăng, giá trị 2 sẽ được trả về:

```
int insertNode(AVLTree &T, DataType X)
{  int   res;
   if(T) {
      if(T->key == X)return 0; //đã có
      if(T->key > X) {
         res= insertNode(T->pLeft, X);
         if(res < 2) return res;
         switch(T->balFactor) {
            case RH: T->balFactor    = EH;
                     return 1;
            case EH: T->balFactor    = LH;
                     return 2;
            case LH: balanceLeft(T); return 1;
         }
      }else {
         res= insertNode(T-> pRight, X);
         if(res < 2) return res;
         switch(T->balFactor) {
            case LH: T->balFactor    = EH;
                     return 1;
```

```
                case EH: T->balFactor   = RH;
                         return 2;
                case RH: balanceRight(T); return 1;
            }
        }
    }
    T  = new TNode;
    if(T == NULL)  return -1; //thiếu bộ nhớ
    T->key= X; T->balFactor = EH;
    T->pLeft = T->pRight = NULL;
    return 2; // thành công, chiều cao tăng
}
```

Hủy một phần tử trên cây AVL

Cũng giống như thao tác thêm một nút, việc hủy một phần tử X ra khỏi cây AVL thực hiện giống như trên CNPTK. Chỉ sau khi hủy, nếu tính cân bằng của cây bị vi phạm ta sẽ thực hiện việc cân bằng lại.

Tuy nhiên việc cân bằng lại trong thao tác hủy sẽ phức tạp hơn nhiều do có thể xảy ra phản ứng dây chuyền. (*Tại sao ?*)

Hàm delNode trả về giá trị 1, 0 khi hủy thành công hoặc không có X trong cây. Nếu sau khi huỷ, chiều cao cây bị giảm, giá trị 2 sẽ được trả về:

```
int delNode(AVLTree &T, DataType X)
{  int   res;
   if(T==NULL) return 0;
   if(T->key > X) {
      res= delNode (T->pLeft, X);
      if(res < 2) return res;
      switch(T->balFactor) {
         case LH: T->balFactor   = EH;
                  return 2;
         case EH: T->balFactor   = RH;
                  return 1;
```

```
            case RH: return balanceRight(T);
         }
      }
      if(T->key < X) {
         res= delNode (T->pRight, X);
         if(res < 2) return res;
         switch(T->balFactor) {
            case RH: T->balFactor      =.EH;
                     return 2;
            case EH: T->balFactor   = LH;
                     return 1;
            case LH: return balanceLeft(T);
         }
      }else { //T->key == X ,
         AVLNode* p = T;
         if(T->pLeft == NULL) {
            T  = T->pRight; res  = 2;
         }else if(T->pRight == NULL){
               T  = T->pLeft; res   = 2;
            }else {  //T có cả 2 con
               res=searchStandFor(p,T->pRight);
               if(res < 2) return res;
               switch(T->balFactor) {
                  case RH: T->balFactor = EH;
                                          return 2;
                  case EH: T->balFactor = LH;
                                          return 1;
                  case LH: return balanceLeft(T);
               }
         }
         delete p;
         return res;
      }
   }
```

//Tìm phần tử thế mạng

```
int searchStandFor(AVLTree &p, AVLTree &q)
{  int   res;
   if(q->pLeft){
      res= searchStandFor(p, q->pLeft);
      if(res < 2) return res;
      switch(q->balFactor) {
         case LH: q->balFactor   = EH;
                  return 2;
         case EH: q->balFactor   = RH;
```

```
                    return 1;
                case RH: return balanceRight(T);
            }
        }else {
            p->key= q->key;
            p  = q;
            q  = q->pRight;
            return 2;
        }
    }
```

NHẬN XÉT

- Thao tác thêm một nút có độ phức tạp O(1).

- Thao tác hủy một nút có độ phức tạp O(h).

- Với cây cân bằng trung bìnhhai lần thêm vào cây thì cần một lần cân bằng lại; năm lần hủy thì cần một lần cân bằng lại.

- Việc huỷ một nút có thể phải cân bằng dây chuyền các nút từ gốc cho đến phần tử bị huỷ trong khi thêm vào chỉ cần một lần cân bằng cục bộ.

- Độ dài đường tìm kiếm trung bình trong cây cân bằng gần bằng cây cân bằng hoàn toàn $\log_2 n$, nhưng việc cân bằng lại đơn giản hơn nhiều.

- Một cây cân bằng không bao giờ cao hơn 45% cây cân bằng hoàn toàn tương ứng dù số nút trên cây là bao nhiêu.

TÓM TẮT

Tiếp nối chương 3, trong chương này chúng ta đã xem xét một dạng cấu trúc dữ liệu động phi tuyến đơn giản và thông dụng: cấu trúc dữ liệu cây. Do nhược điểm về tốc độ truy xuất của danh sách liên kết nên người ta đã tìm kiếm các CTDL khác hiệu quả hơn. Cây là một trong những đáp án cho bài toán này.

Phần đầu của chương trình bày các khái niệm liên quan đến cây nhị phân, các thao tác trên CTDL này và mối quan hệ của nó với cây tổng quát. Phần tiếp theo, cây nhị phân tìm kiếm, một dạng đặc biệt của cây nhị phân đã được giới thiệu. Đây là một CTDL cho phép tìm kiếm hiệu quả hơn hẳn danh sách liên kết. Các thao tác không quá phức tạp nên cây nhị phân tìm kiếm là một trong những CTDL động thông dụng nhất.

Tuy có những ưu điểm rõ rệt so với danh sách liên kết nhưng cây nhị phân tìm kiếm chưa đủ tốt. Trong một số trường hợp, nó bị suy biến thành danh sách liên kết. Điều này thúc đẩy sự ra đời của cây nhị phân tìm kiếm cân bằng (cây AVL). Cấu trúc của cây AVL đảm bảo các thao tác tìm kiếm chỉ tốn chi phí $\log_2 n$ với n là số nút trên cây. Cái giá phải trả là đôi khi xuất hiện nhu cầu cân bằng lại. Việc cân bằng lại sau khi thêm vào một phần tử thường xảy ra hơn là sau khi hủy. May thay, khi thêm một phần tử, ta chỉ phải thực hiện việc cân bằng cục bộ. Trong khi đó, khi hủy một phần tử, việc cân bằng có thể lan truyền lên tận gốc.

Bài tập lý thuyết

45. Hãy trình bày các vấn đề sau đây:

a. Định nghĩa và đặc điểm của cây nhị phân tìm kiếm.

b. Thao tác nào thực hiện tốt trong kiểu này.

c. Hạn chế của kiểu này là gì ?

46. Xét thuật giải tạo cây nhị phân tìm kiếm. Nếu thứ tự các khóa nhập vào là như sau:

8 3 5 2 20 11 30 9 18 4

thì hình ảnh cây tạo được như thế nào ?

Sau đó, nếu hủy lần lượt các nút theo thứ tự như sau :

15, 20

thì cây sẽ thay đổi như thế nào trong từng bước hủy, vẽ sơ đồ (nêu rõ phương pháp hủy khi nút có cả hai cây con trái và phải)

47. Áp dụng thuật giải tạo cây nhị phân tìm kiếm cân bằng để tạo cây với thứ tự các khóa nhập vào là như sau :

5 7 2 1 3 6 10

thì hình ảnh cây tạo được như thế nào ? Giải thích rõ từng tình huống xảy ra khi thêm từng khóa vào cây và vẽ hình

minh họa.

Sau đó, nếu hủy lần lượt các nút theo thứ tự như sau :

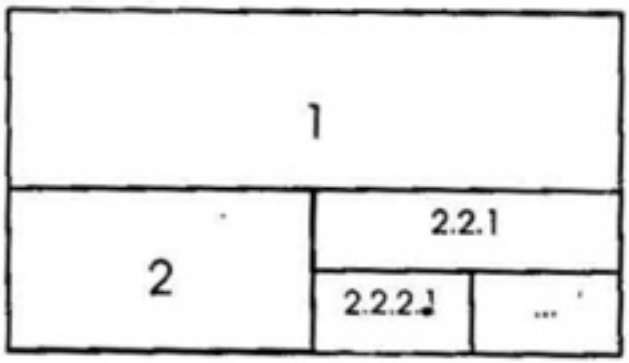
5, 6, 7, 10

thì cây sẽ thay đổi như thế nào trong từng bước hủy, vẽ sơ đồ và giải thích.

48. Viết các hàm xác định các thông tin của cây nhị phân T:

- Số nút lá
- Số nút có đúng một cây con
- Số nút có đúng hai cây con
- Số nút có khóa nhỏ hơn x (giả sử T là CNPTK)
- Số nút có khóa lớn hơn x (giả sử T là CNPTK)
- Số nút có khóa lớn hơn x và nhỏ hơn y (T là CNPTK)
- Chiều cao của cây
- In ra tất cả các nút ở tầng (mức) thứ k của cây T
- In ra tất cả các nút theo thứ tự từ tầng 0 đến tầng thứ h-1 của cây T (h là chiều cao của T).
- Kiểm tra xem T có phải là cây cân bằng hoàn toàn không.
- Độ lệch lớn nhất trên cây. (Độ lệch của một nút là độ lệch giữa chiều cao của cây con trái và cây con phải của nó. Độ lệch lớn nhất trên cây là độ lệch của nút có độ lệch lớn nhất).

49. Cho một hình chữ nhật như hình vẽ, hình chữ nhật này có thể được chia thành hai phần bằng nhau, nếu được chia thành hai phần, các phần này sẽ được đánh số theo thứ tự như trên hình vẽ :



Mỗi phần có thể ở một trong hai trạng thái :

- Trạng thái a: không bị phân chia
- Trạng thái b: tiếp tục phân chia thành hai phần bằng nhau, mỗi phần có thể ở trạng thái a hay b.

Giả thiết sự phân chia là hữu hạn.

a. Tìm cấu trúc dữ liệu thích hợp nhất để biểu diễn hình trên, định nghĩa CTDL đó trong ngôn ngữ Pascal

b. Giả sử đã có một CTDL tương ứng được tạo, viết chương trình in ra danh sách các hình chữ nhật không bị phân chia .

Ví dụ: trong hình vẽ trên, ta có (2.2.2.1), (2.2.1), (2.1), (1)

50. Xây dựng cấu trúc dữ liệu biểu diễn cây N-phân ($2 < N \leq 20$).

Viết chương trình con duyệt cây N-phân và tạo sinh cây nhị phân tương ứng với các khoá của cây N-phân.

Giả sử khóa được lưu trữ chiếm k byte, mỗi con trỏ chiếm 4 byte, vậy dùng cây nhị phân thay cây N-phân thì có lợi gì

trong việc lưu trữ các khoá ?

51. Viết hàm chuyển một cây N-phân thành cây nhị phân.

52. Viết hàm chuyển một cây nhị phân tìm kiếm thành xâu kép có thứ tự tăng dần.

53. Giả sử A là một mảng các số thực đã có thứ tự tăng. Hãy viết hàm tạo một cây nhị phân tìm kiếm có chiều cao thấp nhất từ các phần tử của A.

54. Viết chương trình con đảo nhánh (nhánh trái của một nút trên cây trở thành nhánh phải của nút đó và ngược lại) một cây nhị phân .

55. Hãy vẽ cây AVL với 12 nút có chiều cao cực đại trong tất cả các cây AVL 12 nút.

56. Tìm một dãy N khóa sao cho khi lần lượt dùng thuật toán thêm vào cây AVL để xen các khoá này vào cây sẽ phải thực hiện mỗi thao tác cân bằng lại (LL, LR, RL, RR) ít nhất một lần.

57. Hãy tìm một ví dụ về một cây AVL có chiều cao là 6 và khi hủy một nút lá (chỉ ra cụ thể) việc cân bằng lại lan truyền lên tận gốc của cây. Vẽ ra từng bước của quá trình hủy và cân bằng lại này.

Bài tập thực hành

58. Cài đặt chương trình mô phỏng trực quan các thao tác trên cây nhị phân tìm kiếm.

59. Cài đặt chương trình mô phỏng trực quan các thao tác trên cây AVL.

60. Viết chương trình cho phép tạo, tra cứu và sửa chữa từ điển Anh- Việt.

61. Viết chương trình khảo sát tần suất xảy ra việc cân bằng lại của các thao tác thêm và hủy một phần tử trên cây AVL bằng thực nghiệm. Chương trình này phải cho phép tạo lập ngẫu nhiên các cây AVL và xóa ngẫu nhiên cho đến khi cây rỗng. Qua đó cho biết số lần xảy ra cân bằng lại trung bình của từng thao tác.

TÀI LIỆU THAM KHẢO

1. Th.H. Cormen, Ch. E. Leiserson, R. L. Rivest, *Introduction to Algorithms* (MIT Press, 1998).

2. A.V. Aho, J.E Hopcroft, J.D Ullman, *Data structures and algorithms* (Addison Wesley, 1983).

3. N.Wirth, *Algorithms + Data structures = Programs* (Prentice Hall, 1976)

4. J. Courtin, I. Kowarski, *Initiation à l'algorithmique et aux structures de donneés.* (Tome 1,2,3 _ Dunod , 1990. Bản dịch của Viện Tin học, 1991).

5. Đỗ Xuân Lôi, *Cấu trúc dữ liệu và giải thuật*, (NXB Khoa học và Kỹ thuật, 1996)

6. Robert Sedgewick, *Cẩm nang thuật toán, tập 1* (NXB Khoa học và Kỹ thuật, 1994 - bản dịch của nhóm tác giả ĐHTH TP. HCM)

MỤC LỤC

CHƯƠNG 1 TỔNG QUAN VỀ GIẢI THUẬT VÀ CẤU TRÚC DỮ LIỆU 4

I. Vai trò của cấu trúc dữ liệu trong một đề án tin học 4

II. Các tiêu chuẩn đánh giá cấu trúc dữ liệu 8

III. Kiểu dữ liệu 11
1. Định nghĩa kiểu dữ liệu 11
2. Các kiểu dữ liệu cơ bản 12
3. Các kiểu dữ liệu có cấu trúc 14
4. Một số kiểu dữ liệu có cấu trúc cơ bản 16

IV. Đánh giá độ phức tạp giải thuật 21
1. Các bước phân tích thuật toán 24
2. Sự phân lớp các thuật toán 25
3. Phân tích trường hợp trung bình 28

BÀI TẬP 31

CHƯƠNG 2 TÌM KIẾM & SẮP XẾP 34

I. Nhu cầu tìm kiếm, sắp xếp dữ liệu trong một hệ thống thông tin 34

II. Các giải thuật tìm kiếm nội 35
1. Tìm kiếm tuyến tính 36
2. Tìm kiếm nhị phân 39

III. Các giải thuật sắp xếp nội 42
1. Định nghĩa bài toán sắp xếp 42
2. Các phương pháp sắp xếp thông dụng: 44
3. Phương pháp chọn trực tiếp 45

4. Phương pháp chèn trực tiếp 48
5. Phương pháp đổi chỗ trực tiếp 52
6. Phương pháp nổi bọt (Bubble sort) 57
7. Sắp xếp cây - Heap sort 63
8. Sắp xếp với độ dài bước giảm dần - Shell sort 70
9. Sắp xếp dựa trên phân hoạch - Quick sort 74
10. Sắp xếp theo phương pháp trộn trực tiếp - Merge sort 79
11. Sắp xếp theo phương pháp cơ số - Radix sort 86

BÀI TẬP **93**

CHƯƠNG 3 CẤU TRÚC DỮ LIỆU ĐỘNG **97**

I. Đặt vấn đề **97**

II. Kiểu dữ liệu con trỏ **99**
1. Biến không động (biến tĩnh, biến nửa tĩnh) 99
2. Kiểu con trỏ 100
3. Biến động 102

III. Danh sách liên kết (link list) **104**
1. Định nghĩa 104
2. Các hình thức tổ chức danh sách 104

IV. Danh sách đơn (xâu đơn) **107**
1. Tổ chức danh sách đơn theo cách cấp phát liên kết 107
2. Các thao tác cơ bản trên danh sách đơn 109
3. Sắp xếp danh sách 120
4. Các cấu trúc đặc biệt của danh sách đơn 131

V. Một số cấu trúc dữ liệu dạng danh sách liên kết khác **144**
1. Danh sách liên kết kép 144
2. Hàng đợi hai đầu (double-ended queue) 154
3. Danh sách liên kết có thứ tự (Odered List) 156
4. Danh sách liên kết vòng 157

5. Danh sách có nhiều mối liên kết 161
6. Danh sách tổng quát 162

BÀI TẬP **165**

CHƯƠNG 4 CẤU TRÚC CÂY **172**

I. Cấu trúc cây **172**
1. Một số khái niệm cơ bản 173
2. Một số ví dụ về đối tượng các cấu trúc dạng cây 174

II. Cây nhị phân **175**
1. Một số tính chất của cây nhị phân 176
2. Biểu diễn cây nhị phân T 177
3. Duyệt cây nhị phân 178
4. Biểu diễn cây tổng quát bằng cây nhị phân 182
5. Một cách biểu diễn cây nhị phân khác 183

III. Cây nhị phân tìm kiếm **185**
1. Các thao tác trên cây nhị phân tìm kiếm 186

IV. Cây nhị phân cân bằng (AVL tree) **193**
1. Cây nhị phân cân bằng hoàn toàn 193
2. Cây nhị phân cân bằng 194

BÀI TẬP **210**

TÀI LIỆU THAM KHẢO **215**
MỤC LỤC **217**

GIÁO TRÌNH CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Trần Hạnh Nhi, Dương Anh Đức

NHÀ XUẤT BẢN
ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH
KP 6, P. Linh Trung, Q. Thủ Đức, TPHCM
ĐT: 7242181 + 1421, 1422, 1423, 1425, 1426
Fax: 7242194; Email: vnuhp@vnuhcm.edu.vn

☆☆☆

Chịu trách nhiệm xuất bản
TS HUỲNH BÁ LÂN
Biên tập
NGUYỄN VIỆT HỒNG
NGUYỄN HUỲNH
Sửa bản in
THÙY DƯƠNG
Trình bày bìa
XUÂN THẢO
Đơn vị/Người liên kết:
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

GT. 03. TH(V)
ĐHQG.HCM-10 **484-2009/CXB/118-45** **TH.GT.411-10(T)**

In tái bản 500 cuốn khổ 14.5 x 20.5cm. Số đăng ký kế hoạch xuất bản: 484- 2009/CXB/118-45/ĐHQGTPHCM. Quyết định xuất bản số: 309/QĐ-ĐHQGTPHCM cấp ngày 14/06/2010 của NXB ĐHQGTPHCM. In tại Công ty TNHH In và Bao bì Hưng Phú. In xong và nộp lưu chiểu tháng 7 năm 2010.